定理証明器によって証明された C プログラムのマージャー

後藤 裕貴[1]　高橋 和子[1]

概要：本研究の目的は，ソフトウェア検証過程における定理証明の利用方法の提案と，定理証明の応用領域の拡張である．我々は，ケーススタディとして定理証明器 Isabelle/HOL でシステム仕様を保証した C プログラムマージャーを構築する．これは，与えられた 2 つの C プログラムのソースコードに対し，それらを併合し，いくつかの最適化を施した C プログラムソースコードを出力するマージシステムである．また，ここで扱う C プログラムは実際の C 言語の部分集合である．本システムの主要部分は Isabelle/HOL で記述し証明しており，フロントエンドとバックエンドは C プログラムで実装している．本システムは，フロントエンドとバックエンドを備えることで，定理証明器に不慣れなユーザにも使いやすいツールとして提供可能である．証明では，マージ後のプログラムが変数名の重複を含まないこと，改名された関数定義が必ず存在することなど，構文的な正当性のみを対象とした．マージャーはソフトウェアテストにおけるテストスイートや，分散開発環境でのプログラムソースの併合などで用いられ，仕様を証明したものを提供することで，これを使って開発されたシステムの信頼性が向上する．また，このマージャーを Isabelle/HOL のライブラリとして提供することで，定理証明の応用事例の増加も期待できる．

Certified Merger for C Programs Using a Theorem Prover

Goto Yuki[1]　Takahashi Kazuko[1]

Abstract: The purpose of this study is to propose a manner of a usage of a theorem prover in software verification process and to expand an application area of theorem proving. As a case study, we construct a certified merger for C programs, which is verified using a theorem prover Isabelle/HOL. It is a merge system that generates a merged code with some optimization of a given pair of C programs. Our target is a subset of a real C language. The main part of the system is both written and proved by Isabelle/HOL. The front-end part and the back-end part are implemented in C. It provides a useful tool for users who are not familiar with theorem provers. We prove several lemmas on syntactical correctness. For example, the merged program includes no duplication of variable names, and there exist the corresponding renamed definitions of functions. Mergers are used for a test program of test suits in the process of software verification or in a distributed environment for software development. The certified merger can improve the reliability of these processes. In addition, if we provide our proof as a library of Isabelle/HOL, more usage of theorem provers is promising.

1. はじめに

昨今，ソフトウェアのサイズは大きくなり，その制御構造もより複雑になっている．ソフトウェアを開発する際，いくつかは分散環境において並行して開発されており，組み込みシステムでは，ハードウェアとソフトウェアは統括的に扱われるべきである. これらの事実は，信頼性のあるソフトウェア開発の重要性を示すとともに，コストのかかる困難な作業であることを示唆する．

ソフトウェアの正しさを形式的，また自動的に証明するため，モデル検査や定理証明のような形式手法の導入が提案されている [6]．形式的なアプローチによって、テストベースのアプローチではカバーできないケースも，論理的または数学的に保証することができる．

しかし，形式手法は実際の開発現場ではあまり一般的ではない．その主な理由は，多くのソフトウェア開発者が形

[1] 関西学院大学
Kwansei Gakuin Uniersity

式化になじみがなく，これらのツールの取扱いには高い専門知識と長期の研修が要求されるからである．また，形式手法によるアプローチは，ソフトウェアモデルを厳密に記述する必要があるが，開発者以外のユーザは，内部の複雑な構造と膨大なサイズのため，場合によってはソフトウェアの詳細を分析することができない．例えば，gcc のような商用 C コンパイラの完全な仕様書を，開発者以外が表現することは極めて困難である．そのため現状では，開発における検証段階でシステムテストがまだ実用的であり，頻繁に使われている．

システムテストでは，ソフトウェアプログラムが特定の性質を持っているかを，テストスイート（証明スイート）を使って検証する．テストスイートは，テストケースの集合で，体系化されたプログラム群である [5]．テスト段階での効率を上げるため，分割して格納されている複数のテストケースを実行する際，1 つのプログラムに結合，またはマージして，ファイルのオープン/クローズの時間を削減することがある．ここで，テストプログラムが正しくなければ，テスト時にエラーが発生した場合，証明する対象のソフトウェアと，テストプログラムのどちらが原因か特定できない．よって，テストプログラムの正しさは，証明する対象のソフトウェアより重要である．本発表では，ケーススタディとして，定理証明器を用いて，C コンパイラのための，テストプログラムのマージャーの証明を行う．具体的には，定理証明器 Isabelle/HOL[15] により，C プログラムのマージャーを構築し，その正しさを証明する．また，マージされたプログラムに対し，実行時間短縮とメモリ使用量削減のため，いくつかの最適化を行う．この最適化の正しさに関しても，証明を行う．定理証明技術は，再帰的な構造を記述でき，プログラムモデルを構築しやすい利点と，モデル検査のように，次状態の分岐が多くなることによる状態爆発が起こらない利点があるので，今回のようなマージャーの構築に向いている．

マージャーは複数のプログラムを連結する．連結においては，変数と関数の，名前の重複は排除しなければならない．現在までの，定理証明のプログラムへの応用事例では，こういった構文論はあまり扱われていなかった．今回構築するマージャーの主な働きは，プログラムの意味論よりも構文論の形式化を対象とし，文字列の操作によって改名を行うことである．本研究の目的の 1 つは，構文論を扱う定理証明の応用を探るということである．

マージャーは 3 つの部分から成る．第 1 の部分はフロントエンド部であり，C プログラムのペアが与えられた際，Isabelle/HOL の表現へと変換する．第 2 の部分は主要部であり，変換されたプログラムを，Isabelle/HOL で証明されたマージャーによってマージする．また，Isabelle/HOL で表現されたプログラムに対し，証明された最適化を適用する．第 3 の部分はバックエンド部であり，マージされた結果を C コードに変換する．

Isabelle/HOL が関数型言語に基づくため，プログラミングの様式は C 言語とは異なる．しかし，システムのインタフェースは C プログラムなので，C 言語にさえ慣れているユーザであれば，負担なく使うことができる．さらに Isabelle/HOL をはじめ証明支援系は，前提条件が不十分ならばエラーを報告するという性質から，入力プログラムの構造が正しくなければ検知することができる．この 2 点により，マージャーはシステム開発の効率の向上に貢献することが可能である．このように，本研究のもう 1 つの目的は，定理証明を用いて，実際に開発で使えるようなシステムの提案をすることである．

本発表は，以下のように構成される．第 2 節では，証明支援系 Isabelle/HOL について述べる．第 3 節では，Isabelle/HOL により証明されたマージャーと最適化について述べる．第 4 節では，マージャーのシステムについて説明する．第 5 節では，関連研究を示す．最後に，第 6 節では，結論を提示する．

2. Isabelle/HOL

定理証明器は形式手法のための技術であり，Isabelle/HOL[15]，Coq[4]，ACL2[8]，PVS[16] のようないくつかのツールが開発されている．それらは，証明が完全に自動化されるわけではなく，ユーザーによるナビゲーションが必要となるため，証明支援系と呼ばれることもある．純粋数学から実際のハードウェアやセキュリティプロトコルまで，定理証明が利用された多くのアプリケーションがあり，多くの成功例が報告されている [7]．

ここでは，強力な単純化を行う Isabelle/HOL を用いる．定理証明器の Isabelle は，形式言語を記述，証明するツールとしてケンブリッジ大学の Larry Paulson によって開発された．その後，仕様を高階論理に特化させたものが Isabelle/HOL である．Isabelle/HOL では，データ型や関数は帰納的に定義されるが，その際構文に誤りがあれば受理されない．なお，証明は次のように後ろ向きに行われる．まず最初に，ユーザによって示されたゴールを証明しようとする．そのゴールを直接証明できなければ，定理証明器は仮定などを用いてサブゴールを生成する．このプロセスは各々のサブゴールに対しても繰り返し行われ，すべてのサブゴールが真であると証明できれば，証明は完了する．偽の場合は，証明が完結しないか，あるいは False を出力する．

以下に Isabelle/HOL に関する記述と証明の例を示す．

```
primrec rev :: "'a list ⇒ 'a list" where
"rev [] = []" |
"rev (x # xs) = rev xs @ [x]"


thorem rev_rev[simp] : "rev (rev xs) = xs"
apply auto
done
```

@はリストの連結，#はヘッドとボディからリストを構成する演算子をそれぞれ示す．1 行目の *primrec* は，この関数 *rev* が，原始帰納関数の定義であることを示している．また *'a list ⇒ 'a list* は，リストを引数としてリストを返すことを示している．3 行目は，引数のリストの先頭の要素 *x* をリスト化し，関数 *rev* に引数として残りの要素を与えたものの最後尾に連結している．これは再帰的に実行され，引数として与えるリストが空になったときに 2 行目が実行される．結果として，関数 *rev* は与えたリストの逆順を返す関数となる．

4 行目からの記述は，リスト *xs* に対し，2 度反転させたものは同じリスト *xs* であるという証明である．1 行目には命題を，2 行目には適用させるルールの適用を記述している．Isabelle/HOL に比べて Coq など他の証明支援系は，ルールの適用を細かく指示しなければいけないが，手で証明を行うことと似ており，証明過程を追って理解することは難しくない．Isabelle/HOL は例のように，コマンド *simp* および *auto* によって，証明すべき補題に対し，自動的に適用すべきルールを探し，単純化していく．ライブラリとして定義されているルールも豊富にあるため，この例では *auto* を 1 度適用することで証明が終了する．このとき，どのルールを適用したかは明示されないので，証明過程を追うことは難しい．しかし，自明であるものや，単純な証明に関しては，冗長な記述を行わずに済むことが利点である．

3. Isabelle/HOL によって証明されたマージャーと最適化

3.1 マージャーの概要

C プログラムのペアを扱う際，一方を基本プログラムとし，他方を副次プログラムとする．マージャーは，基本プログラムの構造に副次プログラムの構造を追加することによってマージする．さらに多くのプログラムをマージする場合は，このマージャーを再帰的に用いることによって実現される．

まず副次プログラムについて，メイン関数を含む関数名と，グローバル変数名の先頭部に，ファイル名を追加する．基本プログラムのメイン関数から，改名した副次プログラムのメイン関数を呼び出すようにする．加えて，これらの改名によって影響を受ける要素すべてを改名する．

```
% primary program
int V;
int main(void)
  { V = 1; }
```

```
% secondary program
int V;
int main(void)
  { V = 0; }
```

```
% merged program
int V;
int FV;
int main(void) {
  Fmain();
  V = 1;
}
int Fmain() {
  FV = 0;
}
```

図 1 マージの例

図 1 は例を示している．ここで F は，副次プログラムのファイル名である．図 1 では，どちらのプログラムもグローバル変数 V を含んでいる．副次プログラムのグローバル変数名は FV に改名され，またメイン関数名も Fmain に改名される．ここで，仮に基本プログラムが FV を含んでおり，副次プログラムに V があれば，重複が発生してしまう．

3.2 最適化の概要

マージされた結果に対して，最適化を行う．ここでは 2 つの最適化を定義している．1 つ目は，参照されていない変数（デッドコード）の削除である．2 つ目は，インライン展開である．最適化は他にも，共通式削除やレジスタ割り当てなど多く存在するが，このシステムでは構文を対象にし，比較的適用しやすい項目に限定して実装している．また，最適化の正しさについていくつかの補題を証明している．

3.3 形式化

C プログラムのペア, マージャー, 最適化を, Isabelle/HOL により形式的に表現する．もし変数や関数の型構造に矛盾がある場合，その定義は受理されない．定義が受理されるならば，基本的に構文的な正しさは保証される．マージャー，最適化はいずれも複数の入力プログラムを構文的に変換し，新しいプログラムを出力するものであり，文字列の操作が基本である．

出力されたプログラムにおいて変数に重複がないことと，呼出し元と呼ばれるものの対応関係が保たれているかを証明する必要がある．

ライブラリに文字列を扱うものは既にあるが，今回は扱わない”Nibble”のような定義があることや，証明が複雑に

なるという点から，このライブラリは用いていない．今回は文字列を単純に扱うため，新たなモデルとして最低限の文字列を定義する．文字列は，アルファベットの大文字を 1 つの文字としたリストで表現する．また，文字列の連結は，リストの連結関数を用いて実現する．

ここで，C プログラムは，FileName，GvarList，Funcs，Main の 4 つ組で構成され，Prog というデータ型で定義される．

$$Prog = "FileName * GvarList * Funcs * Main"$$

FileName はプログラムが書かれたファイル名である．GvarList はプログラムで宣言されているグローバル変数のリストである．グローバル変数は，変数の型と変数名で構成されている．Funcs は，関数定義のリストである．各関数は，戻り値の型，関数名，引数のリスト，文のリストとして構成されている．Main はメイン関数である．

Isabelle/HOL は形式言語に基づいているので，演算モデルは帰納的に定義でき，入力タームは出力タームへ書き換えられる．

以降 V1 + V2 は，変数 V1 と V2 の文字列を連結したものとし，F は副次プログラムの FileName とする．

merge は merge_func，merge_gvar，add_change_main1，add_change_main2 の，4 つの関数で構成される．merge に含まれるすべての関数は，Prog から Prog への関数である．

$$\begin{aligned}&fun\ merge :: "Prog \Rightarrow Prog \Rightarrow Prog"\\&where "merge\ pr1\ pr2 =\\&(merge\_func(merge\_gvar\ (add\_change\_main1\\&(add\_change\_main2\ pr1\ pr2)\ pr2)\ pr2)\ pr2)\end{aligned}$$

pr1，pr2 は関数 merge の 2 つの引数となるプログラムである．pr2 の Main において，(add_change_main2 pr1 pr2) は各関数呼び出し CF を F+CF，各グローバル変数 GV を F+GV に変更し，新たな関数の関数本体とし，その関数名を F+main として pr1 の Funcs に追加したものを結果のプログラムとして出力する．

(add change main1 pr1 pr2) は，pr1 の Main に F+main の関数呼び出しを追加したものを結果のプログラムとして出力する．

(merge_gvar pr1 pr2) は，pr2 の GvarList の各要素名 VN を F + VN とし，pr1 の GvarList に追加したものを結果のプログラムとして出力する．

(merge_func pr1 pr2) は，pr2 の Funcs の各関数について，その関数名 FN を F+FN に変更し，関数本体に各関数の呼び出し CF があれば，それを CF から F+CF に変更し，pr1 の Funcs に追加したものを結果のプログラムとして出力する．

$$\begin{aligned}&fun\ change\_funcs :: "Prog \Rightarrow Funcs \Rightarrow Funcs"\\&\ where "change\_funcs\ pr\ xs =\\&callfunc\_changename2\ pr\ (funcs\_changename\\&pr\ xs)"\end{aligned}$$

この例は，関数定義を示している．関数は Prog と Funcs を入力として与えられ，Funcs を出力するもので，funcs_changename と callfunc_changename2 という関数から構成される．

多くの定義を含むデータ型もリストとして扱うことができる．このとき，高階関数を利用すると，ライブラリ有効利用でき，定義や証明が簡潔になる．

$$\begin{aligned}&fun\ gvar\_changename :: "Prog \Rightarrow GvarList \Rightarrow\\&GvarList"\ where "gvar\_changename\ pr\ gs =\\&map\ (\lambda x.\ case\ x\ of\ (GV\ gvtype\ gvname) \Rightarrow\\&(GV gvtype\ ((fst\ pr)\ @\ gvname)))\ gs"\end{aligned}$$

この例では，無名関数は GvarList gs の各要素である (GV gvtype gvname) にファイル名である (fst pr) を追加している．また関数 map は GvarList のすべての要素に無名関数を適用する．

以上がマージモデルを説明したものである．

最適化については，デッドコードの削除とインライン展開の，2 つの定義を示す．デッドコードの削除は，以下のように定義されている．

$$\begin{aligned}&fun\ reduce\_GvarList\ ::\\&"Prog \Rightarrow GvarList \Rightarrow GvarList"where\\&"reduce\_GvarList2\ pr\ [] = []"|\\&"reduce\_GvarList2\ pr\ (g\#gs) =\\&(if\ g \in set\ (Prog2ExpGvarList\ pr)\\&then\ g\#(reduce\_GvarList2\ pr\ gs)\\&else\ (reduce\_GvarList2\ pr\ gs))"\end{aligned}$$

関数 reduce_GvarList は，Prog と GvarList を引数とし，GvarList を返す．GvarList の各要素に対し，Prog に出現する式に含まれるかどうかを探索し，式に含まれている要素があれば，その要素を新たなリストに追加する. 式になければ追加を行わない．この操作が終われば次の要素に対し同じ操作を繰り返し行う再帰関数である．最終的に構成されたリストを新たな GvarList とすることで，Prog の式中に存在しないものは削除された GvarList を構成できる．

また，インライン展開は以下のようになる．

```
fun inlining_Func :: "Statement ⇒ Name ⇒ Prog ⇒ StatementList" where
"inlining_Func(If exp st1 st2) fn pr =
[(If (change_ExpCallLvarName fn exp) st1 st2)]"|
"inlining_Func(While exp st) fn pr =
[(While (change_ExpCallLvarName fn exp) st)]"|
"inlining_Func (Returnexp) fn pr =
[(Return (change_ExpCallLvarName fn exp))]"|
"inlining_Func (CallFunc name argl) fn pr =
concat (map (λx.(inlining_Funcx3 x name pr))
(Func_StatementList
(find_Func(Prog_Funcs;pr) name)))"|
"inlining_Func(Subs lhs exp) fn pr =
[(Subs lhs (change_ExpCallLvarName fn exp))]"|
"inlining_Func(Dec(LV type name)) fn pr =
[(Dec(LV type(fn@name)))]"|
"inlining_Func(DecA(LV type name) n) fn pr =
[(DecA(LV type fn@name) n)]"
```

関数 inlining_Func は，関数の定義の文において，それが関数呼出し以外であれば，式中のローカル変数 Lvar の名前の先頭に関数名を付加し，関数呼出しであれば，呼び出されている関数の文に置き換える．置き換えのため，別に定義した find_Func を用いて，関数呼出しの関数名と同名の関数を探索している．Lvar を改名しているのは，置き換えをすることによって，Lvar のスコープ範囲が変わり，他の変数と名前が重複する可能性があるからである．

3.4 マージの正しさに関する証明

マージの正しさとして，改名が一貫して行われていること，結果のプログラムで変数名と関数名に重複がないこと，呼ぶものと呼び出されるものの関係性が失われていないことについて証明する．これらの点を記述したいくつかの補題を示す．

以下は，基本プログラムと副次プログラムのそれぞれが，グローバル変数名で重複を持たず，かつ，副次プログラムに V という変数名のグローバル変数があったとき，F+V という変数が基本プログラムに存在せず，かつ，副次プログラム名が空でないならば，マージ後のプログラムには重複がないという補題である．

```
lemma no_duplication :
(distinct (Prog2GvStrl pr1) ⋀
distinct(Prog2GvStrl pr2) ⋀(∀a.((a ∈
set (Prog2GvStrl pr1)) ⟶ (hd a ≠
hd (fst pr2))))fst pr2 ≠ [] ⟹
distinct (Prog2GvStrl (merge_gvar pr1 pr2))
```

ここでは ⟶ は高階論理で使われる論理的接続を意味する．また ⟹ は推論規則を意味する．当初，我々は条件を見逃していたため，エラーが発生したが，後に条件を追加することによって，証明に成功した．

以下は，副次プログラムにおいて，グローバル変数の改名は正しく行われているという補題である．つまり，変更後の変数名は，その先頭にファイル名が追加されているということである．

```
lemma correct_renaming_of_gvs :
Prog_GvarList(merge pr1 pr2) =
(Prog_GvarList pr1)@(gvar_changename pr2
(Prog_GvarList pr2))
```

以下は，マージ後のプログラムの main 関数の先頭は，F+main 関数の呼出しであるという補題である．

```
lemma correct_trans_of_main :
(hd(Func_StatementList(Prog_Main
(merge pr1 pr2)))) =
(CallFunc((fst pr2)@(Func_Name(Prog_Main
pr2))) [])
```

以下は，マージ後のプログラムにおいて呼び出される F+main 関数は，Funcs で定義されているという補題である．

```
lemma correct_trans_of_body :
(Statement2CallFuncName(hd
(Func_StatementList(Prog_Main(merge pr1 pr2)))))
∈ set(Prog2FuncNamel (merge pr1 pr2))
```

Isabelle/HOL によるマージャーは，定義や補題を含め約 535 行であり，補題の数は 11 である．

3.5 最適化

最適化の正しさとして，最適化後に，最適化による削除が反映されているかと，同じスコープ内の変数には名前の重複が起こらないことの 2 つを補題として記述した．

以下は，任意の Gvar は，プログラム中の式において出現しないならば，デッドコード削除後の GvarList にも含まれないという補題である．

```
lemma "∀gv.(¬(find_ExpGvar_in_Prog pr gv)) ⟹
gv ∉ set(Prog_GvarList(reduce_GvarList pr))"
```

以下は，メイン関数で宣言されている Lvar に重複がなければ，インライン展開後のメイン関数で宣言されている Lvar にも重複がないという補題である．

$lemma"(distinct(Prog2MainDecLvarNamel\ pr) \bigwedge distinct(Prog2FuncNamel\ pr) \bigwedge (\forall a.((a \in set(Prog2FuncNamel\ pr)) => (hd\ a \neq hd(Prog2MainDecLvarName\ pr)))))) \Longrightarrow distinct(Prog2MainDecLvarNamel(inlining\ pr))"$

4. システム

4.1 システム構成

証明された C プログラムのマージャーは，フロントエンド部，主要部，バックエンド部（図 2）の 3 つの部分から構成される．フロントエンド部は，C プログラムのペアを Isabelle/HOL の表現に変換する．主要部は, Isabelle/HOL で記述された証明されたマージャーである．ここで, いくつかの最適化も適用される．バックエンド部は, Isabelle/HOL で表現されたマージ結果を C プログラムに変換する．

4.2 対象プログラムの表現

対象とする C プログラムは完全な C プログラムの部分集合である（図 3）．それは，代入文や関数呼出しなど必要最小限の要素を包含するが, 配列や構造体，ポインタ，”goto”や”break”などの文は扱わない．その理由の 1 つは，この研究の目的が Isabelle/HOL を用いてマージャーを作成する有効性を示すことだからである．また，もう 1 つの理由は，このケーススタディのマージャーが，基本的な要素から成るテストプログラムを対象としているからである．

4.3 C プログラムから Isabelle/HOL への変換

C プログラムは，宣言部，メイン関数，関数のリストの 3 つの部分から構成される．フロントエンド部は，C のソースコードを読み，グローバル変数の宣言や，副次プログラムの関数定義のリストに含まれる各関数について，関数の定義を，対応する配列に保存する．そして，それらを Isabelle/HOL のプログラムデータに変換する．図 4 は，フロントエンド部のプロセスのイメージを示している．

4.4 Isabelle/HOL から C プログラムへの変換

Isabelle/HOL は，基本的に与えられた補題を証明して，結果として真か偽かを導出する．ここで我々は，グローバル変数および関数定義のリストと対応する変数の結果として生じる値を得るために，”value”コマンドを用いて評価を行う．Isabelle/HOL は，”Proof General”とよばれる Emacs インタフェースで処理され，Emacs コマンドを用いて結果をファイルに書き出すことができる．そのデータを配列に格納し，結果を C コードとして生成されるようにした．最後にマージされた C プログラムを得る．

5. 関連研究

コンパイラ検証のための，定理証明技術の応用の始まりは，Moore の初期の研究に遡る [14]．その後，コンパイラ検証のいくつかの研究があった．Leinenbach は，Isabelle/HOL の Hoare 論理を用いて，C の部分集合である C0 コンパイラの正しさを示した [9]．また彼らはハードウェアからソフトウェアを統合的に扱って証明する Verisoft プロジェクトで研究を行った．Leroy は Coq を用いて証明された C コンパイラを構築しており [10]，実用に近い研究だった．そのシステムは，C の大きなサブセットである Cminer を入力とし，いくつかの中間言語を通して機械語に変換する．彼らは生成したコードの正しさを証明し，その後は，変換のプロセスで証明を要する部分について研究を進めている [2][11][12]．Strecker は，Isabelle/HOL で書かれた Java コンパイラの証明を行った [17]．また，バックエンド部が正しいコードを生成することも証明している [3]．Zimmerman は，バックエンド部のシステム仕様を，抽象状態機械 (ASM) に基づいて書き直すことによる証明アプローチを提示した [19]．このアプローチは，正しいコンパイラを生成するため，PVS で実行される．また彼らは，中間言語から機械語までの変換の正しさを証明した．彼らの主要な目的は，現実のプロセッサのためにコードを生成するコンパイラを提供することにある．その証明では，PVS が使われているが，手で証明された部分も含まれる．Mansky は，グラフ表現で定められる最適化器の正しさを証明した [13]．

これらの多くの研究は，意味論的な視点でコンパイラの正しさを証明する．他方，我々はコンパイラ自身ではなく，テスト段階に使われるソフトウェアの正しさを構文論的な視点で証明するという，別のアプローチをとった．

6. まとめ

我々は, Isabelle/HOL によって証明された C プログラムのマージャーを開発した．また，いくつかのサンプルプログラムをマージャーに適用し，各々のプログラムと，マージ結果のプログラムが同義であることを確認した．入力プログラムにおいて，機能が十分に定義されていなかったり，いくらかの状態を見逃していても，大抵のマージャーはそのケースを検知しないが，証明されたマージャーは，その際にエラーを報告することが長所である．定理証明の利用によって，入力プログラムが満たさなければならない条件を明確にすることができる．これはシステム開発支援として定理証明が適用される理由の 1 つである．また，C プログラムによるインタフェースは，定理証明に詳しくないユーザでも容易に使用できる．これにより，例えば開発したソフトウェアプログラム，あるいはそのテストの際に

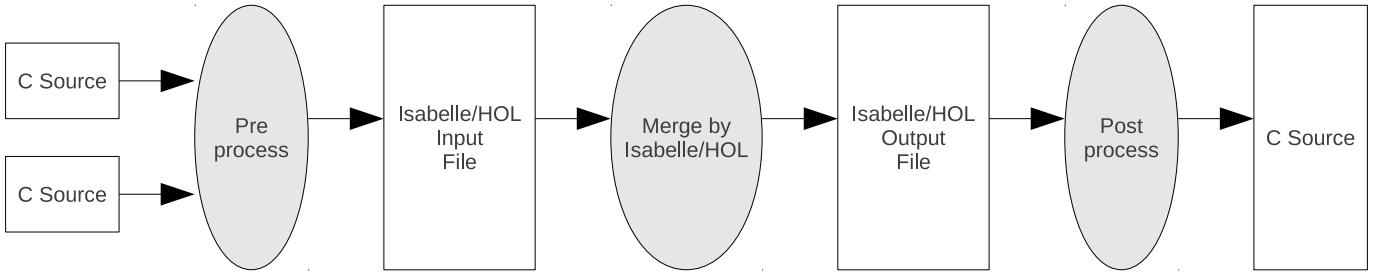

図 2 システム構成

$program ::= (defVar";") * (decFun";") * defMain(defFun)*$
$defVar ::= type\ varName$
$decFun ::= type\ funName"("("void"|decVar(","decVar)*)")"$
$defFun ::= type\ funName"("("void"|decVar(","decVar)*)")"funBody$
$defMain ::= "int""main""(void)"funBody$
$funBody ::= (stmt)*$
$type ::= "int" \mid "double"$
$stmt ::= E|funCall";"|assStmt|decVar$
$assStmt ::= expLhs" = "exp";"$
$Lhs" = "exp";"$
$exLhs ::= varName$
$varName ::= CHAR(CHAR)*$
$exp ::= exp2$
$exp2 ::= exp3$
$exp3 ::= exp4$
$exp4 ::= exp5$
$exp5 ::= INT \mid CHAR \mid funCall \mid refVar$
$refVar ::= varName$
$funCall ::= funName"("argList")"$
$funName ::= CHAR(CHAR)*$
$argList ::= E|exp(","exp)*$
$CHAR ::= "A" \mid "B" \mid "C" \mid "D" \mid "E" \mid "F" \mid "G" \mid "H" \mid "I" \mid "J" \mid "K" \mid "L" \mid "M" \mid "N" \mid "O" \mid "P" \mid "Q" \mid "R" \mid$
$"S" \mid "T" \mid "U" \mid "V" \mid "W" \mid "X" \mid "Y" \mid "Z"$
$INT ::= "0" \mid "1" \mid "2" \mid "3" \mid "4" \mid "5" \mid "6" \mid "7" \mid "8" \mid "9"$

図 3 対象とするプログラムの構文

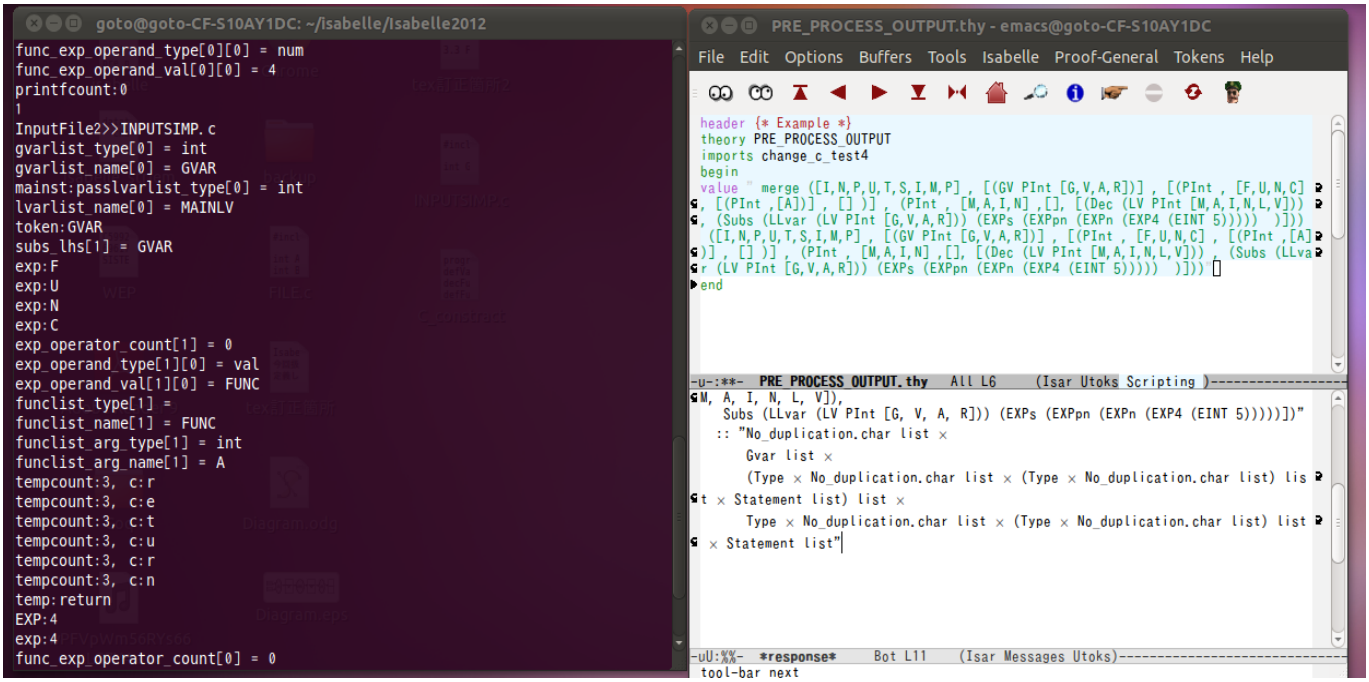

図 4 フロントエンド部プロセスのスクリーンショット

入力するプログラム群において，証明されたマージャーを利用することで，実行における時間効率の向上が期待できる．加えて，それは検証の対象となるソフトウェア開発者以外も使用でき，入力プログラムの構造に不備があれば事前の検知も可能である．

現在，対象とする C プログラムには制約がある．我々は，これを緩和することを最優先に考えている．また，フロントエンド部やバックエンド部についても証明することが必要である．加えて，意味論も考慮する場合は，異なるモデルを定義しなければならない．最適化の正しさを示す補題の証明を完成することも課題である．

今後は，まずマージの入出力や，最適化の入出力を比較し，各文の構成が正しく対応しているという性質を補題として記述し，証明することを目指す．

参考文献

[1] G. Barthe, D. Demange and D. Pichardie: *A formally verified SSA-based middle-end Static Single Assignment meets CompCert*, ESOP12, pp.47-66, 2012.

[2] G. Barthe, D. Demange and D. Pichardie: *Extracting a formally verified, fully executable compiler from a proof assistant*, COCV03, pp.33-50, 2003.

[3] Y. Bertot and P. Castéran: *Interactive Theorem Proving and Program Development: Coq'art- The Calculus of Inductive Costructions*, Springer, 2004.

[4] C. Cheng: *The test suite generation problem: Optimal instances and their implications*, Discrete Applied Mathematics ,Vol.155, No.15, pp.1943-1957, 2002.

[5] E. Clarke and W. Jeannette: *Formal methods: state of the art and future directions*, ACM Computing Surveys, Vol.28, No.4, pp.626-643, 1996.

[6] H. Geuvers: *Proof assistants: History, ideas and future, Sadhana : Academy Proceedings in Engineering Sciences* , Indian Academy of Sciences, Vol.34, No.1, pp.3-25, 2009.

[7] M. Kaufmann, P. Monolios and J. Moore: *Computer-Aided Reasoning: An Approach* , Indian Academy of Sciences, Kluwer Academic Publishers, 2000.

[8] D. Leinenbach, W. Paul and E. Petrova: *Towards the formal verification of a C0 compiler: Code generation and implementation correctness* , SEFM05, pp.2-12, 2005.

[9] X. Leroy: *Formal certification of a compiler back-end or: programming a compiler with a proof assistant* , POPL06, pp.42-54, 2006.

[10] X. Leroy: *Formal verification of a realistic compiler*, Communications of the ACM, Vol.52, No.7, pp.107-115, 2009.

[11] X. Leroy: *A formally verified compiler back-end*, Journal of Automated Reasoning, Vol.42, No.4, pp.363-446, 2009.

[12] W. Mansky and E. Gunter: *A framework for formal verification of compiler optimizations*, ITP10, pp.371-386, 2010.

[13] J. Moore: *A mechanically verified language implementation*, Journal of Automated Reasoning, Vol.5, pp.461-492, 2009.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel: *Isabelle/HOL A Proof Assistant for Higher-Order Logic*, Springer, 2008.

[15] S. Owre, J. Rushby and M. Shankar: *PVS: A prototype verification system*, CADE92, pp.748-752, 1992.

[16] M. Strecker, J. Rushby and M. Shankar: *Formal verification of a Java compiler in Isabelle*, CADE02, pp.63-77, 2002.

[17] W. Zimmermann and T. Gaul: *On the construction of correct compiler back-ends: an ASM-approach*, Journal of Universal Computer Science, Vol.3, No.5, pp.504-567,

1997.