書き換え規則に基づくAPIベース言語拡張のためのCOINSを用いたフレームワークの設計と実装

塩出 拓也[1,a)] 川端 英之[1,b)] 北村 俊明[1,c)]

概要：アプリケーションの開発において，ライブラリの利用は不可欠である．ライブラリの提供する機能は，特殊なアルゴリズムやデータ構造を利用するもの，ネットワークや外部デバイスを用いるものなど様々なものが挙げられる．ライブラリを用いるにはソースコードに API 呼び出しを記述するが，多数の引数を持つ API 呼び出しや，ライブラリの提供する特殊なデータ構造の初期化／終了処理の挿入は一般的に煩雑な作業であり，アプリケーション開発者に過度な負担を強い得る．これに対し，より簡便な記述のプログラムから API 呼び出し系列の制御を行う複雑なプログラムを生成できるコンパイラがあれば，アプリケーション開発者の負担は大幅に緩和される．しかしながら，コンパイラの開発は多くの場合，多大な手間を要する．本稿では，ライブラリの利用を容易にするコンパイラの作成を支援するために，書き換え規則に基づいてコンパイラを構築できるフレームワークを提案する．本システムは，抽象構文木ベースのパターンマッチと編集操作（挿入，置換，削除）によりプログラムの書き換えを実現する．記号表の参照や編集も可能であり，変数の型の参照や新たな変数の宣言を行うことができる．開発したプロトタイプでは，C 言語のプログラムを入力として，ソースコードの一部を API 呼び出しに置き換えた C 言語のプログラムを出力する．実装には COINS コンパイラ・インフラストラクチャを活用している．COINS は HIR というソースコードと同等の制御構造である中間表現を持っており，HIR から HIR への変換モジュールを作成することでプログラム変換を実現した．本システムの適用事例として，MPFR ライブラリ用の書き換え規則を作成した．本システムを用いることで書き換え規則からコンパイラを自動生成し，MPFR ライブラリを意識することなくプログラミングが行える環境を容易に構築できることを確認した．

1. はじめに

アプリケーションの開発において，汎用性の高いプログラムの再利用，すなわちライブラリの利用は不可欠である．ライブラリの提供する機能は，特殊なアルゴリズムやデータ構造を利用するもの，ネットワークや外部デバイスを用いるものなど様々なものが挙げられる．これらの機能は，ライブラリの API 呼び出しをプログラムに記述することで利用できる．

ライブラリを用いることで，アプリケーション開発者はアルゴリズムの実装を意識せずとも，ライブラリの機能を利用することができる．一方で，ライブラリを使うために API 呼び出しをプログラムに記述する作業は煩雑でもある．例えば，任意精度浮動小数点を提供する MPFR ライブラリを用いることを考えよう．図 1(a) に示す C 言語のプログラムは，倍精度浮動小数点型では精度が不足しており期待する結果を得ることができない例である．図 1(a) に示すプログラムを MPFR ライブラリを用いた演算に書き換えたプログラムを図 1(b) に示す．図 1(a) のプログラムを図 1(b) のプログラムに書き換えるためには，次に示す事項を意識する必要がある．

- API 呼び出しにおける引数の順序やデータ型を把握した上で，正確なプログラムの記述
- ライブラリの提供する特殊なデータ構造（例：MPFR ライブラリにおける mpfr_t 型）の初期化／終了処理の挿入
- 複雑な式を二項演算の形に分解すること

このようにアプリケーション開発者は様々なことを考えながら書き換えを行う必要があるため，ライブラリの利用は必ずしも容易とはいえない．

ライブラリの利用に伴う API 呼び出しの手間を軽減する方法はいくつか考えられる．C 言語のマクロ定義を用いることで，定型的な API 呼び出しの記述の集約や，擬似的な構文の追加が可能である．しかしながら，マクロ定義による置換処理の効果は，API 呼び出しの記述量が減るのみであり本質的な解決にはならない．他には，オペレーターオーバーローディングを用いて API 呼び出しを隠蔽する

[1] 広島市立大学 〒 731-3194 広島市安佐南区大塚東 3-4-1
[a)] shiode-lm@hiroshima-cu.ac.jp
[b)] kawabata@hiroshima-cu.ac.jp
[c)] kitamura@hiroshima-cu.ac.jp

```c
#include <stdio.h>
int main(){
  double a,b,c;
  a=1.000000000000001;
  b=1.000000000000000;
  c=(a-b)*1e15;
  printf("%.30lf\n",c);
}
```

(a)計算誤差の発生する
C言語のプログラム

```c
#include <stdio.h>
#include <mpfr.h>
int main(){
  mpfr_t mpfr_a;
  mpfr_t mpfr_b;
  mpfr_t mpfr_c;
  mpfr_t mpfr_t0;
  mpfr_t mpfr_t1;
  mpfr_init2(mpfr_a,113);
  mpfr_init2(mpfr_b,113);
  mpfr_init2(mpfr_c,113);
  mpfr_init2(mpfr_t0,113);
  mpfr_init2(mpfr_t1,113);
  mpfr_set_str(mpfr_a,"1.000000000000001",10,0);
  mpfr_set_str(mpfr_b,"1.000000000000000",10,0);
  mpfr_sub(mpfr_t0,mpfr_a,mpfr_b,0);
  mpfr_set_str(mpfr_t1,"1e15",10,0);
  mpfr_mul(mpfr_c,mpfr_t0,mpfr_t1,0);
  printf("%.30lf\n",mpfr_get_d(mpfr_c,0));
  mpfr_clear(mpfr_a);
  mpfr_clear(mpfr_b);
  mpfr_clear(mpfr_c);
  mpfr_clear(mpfr_t0);
  mpfr_clear(mpfr_t1);
}
```

(b)MPFRライブラリを用いた
C言語のプログラム

図 1 プログラム例

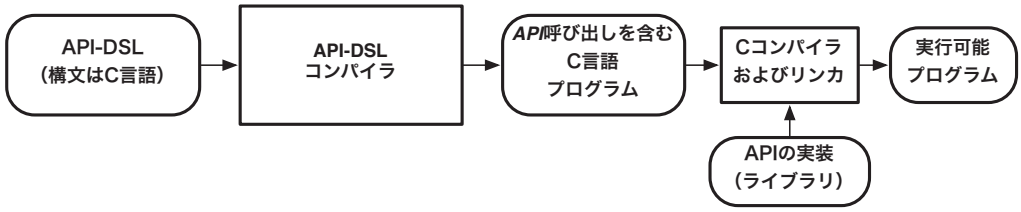

図 2 API-DSL コンパイラとその処理フロー

方法が考えられる．例えば，任意精度浮動小数点演算を提供する MPFR C++ライブラリ [1] が提供する mpreal クラスがある．mpreal クラスは，C++言語標準の演算子を用いた演算の記述が可能であるため，ライブラリの利用において煩雑な API 呼び出しの記述を必要としない．ただし，mpreal 型は変数を文字列で初期化しなければ精度向上の効果を得られないことや，mpreal 型の変換には API 呼び出しを用いた明示的なキャストが必要なことなど，いくつかの注意点がある．結果的にオペレーターオーバーローディングにより記述量の削減にはなるが，ライブラリの仕様や制約を意識してプログラムを記述することに変わりはない．

API が提供された拡張環境におけるプログラミングでは，プログラミング言語自体の役割は API 呼び出し系列の制御である．API の提供によって拡張された計算機環境を「API マシン」と呼ぶことにすれば，ここで得られるプログラムは API マシンをターゲットとするマシン語記述に他ならない．API マシンのためのマシン語記述にあたる複雑なプログラムをより簡便な文法の DSL（Domain Specific Language）から自動生成できれば．アプリケーション開発者の負担は大幅に削減される．

API マシンのための DSL（以下，API-DSL）は API マシン専用のプログラミング言語である．すなわち，アプリケーション開発者は新しいプログラミング言語を習得する必要があり，ライブラリの API 呼び出しをプログラムに記述するよりも大きな負担となりうる．ここで，API-DSL として C 言語の構文をそのまま利用することを考えよう．自然な記述がなされた C 言語のプログラムを入力として，API 呼び出しの系列に置き換えられたプログラムを出力する API-DSL コンパイラがあれば，C 言語の構文を API-DSL として利用できる（図 2）．アプリケーション開発者の記述した C 言語のプログラムは部分的に API マシンのコードに変換されるため，本来の C 言語とは異なる振る舞いになることもありうる．しかし，この振る舞いの差異が自然な読み替えで通じる範囲であれば，アプリケーション開発者は自然にプログラムを記述できる．例えば，MPFR ライブラリでは図 1(a) の C 言語のプログラムを API-DSL と仮定する．そして，MPFR ライブラリ用の API-DSL コンパイラは図 1(a) のプログラムから図 1(b) のプログラムを自動生成する．この方法により，アプリケーション開発者は MPFR ライブラリの仕様や制約を意識せずに，MPFR ライブラリを用いたアプリケーションを開発することがで

きる．

API-DSL コンパイラを構築する方法はいくつかある．コンパイラを新規に作成するのであれば，パーザや中間表現の作成などに多大な手間が必要となってしまう．より効率的な開発をするために，ソースコードが公開されておりユーザが自由に手を加えることができるコンパイラをベースにして API-DSL コンパイラとして構築する方法が考えられる．先行研究 [2] では，コンパイラインスラストラクチャ COINS[3] を活用して MPFR ライブラリ用の API-DSL コンパイラを実装している．先行研究の手法では 1136 行のソースコード追加で実装できたが，COINS のソースコードは 30 万行以上あるため内部構造を十分に理解して拡張するのは容易とはいえない．

そこで我々は，API-DSL コンパイラの開発を容易にするために，書き換え規則に基づく API ベース言語拡張のためのフレームワークを提案する．提案するフレームワークを用いることで，API-DSL コンパイラ開発者は書き換え規則を記述するだけで API-DSL コンパイラを開発することができる．書き換え規則の記述に際して API-DSL コンパイラ開発者に求めるのは，プログラムがコンパイラの内部表現に変換された後の抽象構文木をイメージできることだけである．書き換え規則を記述するための簡単なマニュアルといくつかサンプルを理解するだけで，API-DSL コンパイラ開発者が API-DSL コンパイラを構築できるようになることを目標とする．

提案するシステムでは，抽象構文木ベースのパターンマッチを用いたプログラムの書き換えを行う．抽象構文木ベースにすることで，複雑な式に対するパターンの表現が容易になることや，型やスコープルールなどの情報を活用できる利点がある．

提案するシステムの実現にあたり，本研究ではコンパイラインフラストラクチャ COINS を活用する．COINS は C 言語プログラムを中間表現に変換するフロントエンドと，中間表現を C 言語のプログラムに変換して出力するモジュールを持っている．COINS を活用することにより，書き換え規則に従って中間表現の変換モジュールを開発することで，提案するシステムを実現できる．

以下，2 章で実装に活用する COINS について述べる．3 章では，書き換え規則に基づく API ベース拡張言語のためのフレームワークの設計について述べ，4 章で適用事例の紹介を行い，5 章で評価と考察，6 章で関連研究との比較，7 章でまとめについて述べる．

2. COINS

COINS[3] は，コンパイラの研究，開発，教育およびコンパイラ技術の蓄積を容易にすることを目的として開発されたコンパイラ・インフラストラクチャである．COINS はコンパイラを構成する基本機能のモジュールをすべて備えており，それらの組み合わせを変えたり，一部のモジュールを新たに開発するだけで新しいコンパイラを実現することができる．各モジュールはソース言語やマシン言語および中間表現を入出力するといった共通のインタフェースを提供している．

COINS は高水準中間表現（以下，HIR）と低水準中間表現（以下，LIR）を持っている．HIR はソースコードと同等な制御構造を持っているので，ソースコードに近い形でプログラムの書き換えが可能である．LIR は抽象レジスタマシンに対するアセンブリ言語の形態をとっており，機械語レベルでの最適化処理を可能とする．プログラムで宣言された変数や構造体などの情報は，2 つの中間表現とは別にシンボルテーブルで管理している．提案するシステムはソースコードと同等な抽象構文木に対するパターンマッチを行うため，HIR 上でプログラムの変換を実現する．

他のコンパイラインフラストラクチャに LLVM[4] がある．LLVM は，BitCode という SSA 形式の中間表現を用いている．SSA 形式は，各変数が一度のみ代入されるという制約を持つ中間表現であり大域的な最適化に向いている．その反面，ソースコードに現れる制御構造は保たれておらず，抽象構文木レベルでのパターンマッチを行うのには向いていない．この理由から，提案システムの開発に活用するコンパイラインフラストラクチャとして COINS を用いることにした．

3. 書き換え規則に基づく API ベース拡張言語のためのフレームワークの設計

3.1 システムの全体像

提案するシステムの全体像を図 3 に示す．提案するシステムは，API-DSL プログラムを書き換え規則に従って API 呼び出しを含む C 言語のプログラムに変換する．変換された C 言語のプログラムは，C 言語のコンパイラを用いてコンパイルを行うことで実行可能プログラムを生成できる．本研究で開発するのは，図 3 の二重太枠で示される API マシンのための HIR 変換モジュールである．

3.2 書き換え規則

コンパイラにおいて，プログラムの最適化や変換を行う機能は抽象構文木に対するパターンマッチと置換により実現されている．抽象構文木に対するパターンマッチは，高水準言語で表現できる範囲であらゆるプログラムの書き換えを実現できる．しかしながら，抽象構文木を直接的に編集する機能の開発において，開発者はコンパイラの内部構造について熟知する必要がある．また，抽象構文木の整合性を保ちながら変換することは本来の目的とは異なる部分に手間がかかってしまうため，コンパイラに直接手を入れることは容易ではない．そこで，提案するシステムにおける書き換え規則は，コンパイラの内部構造に対する知識を

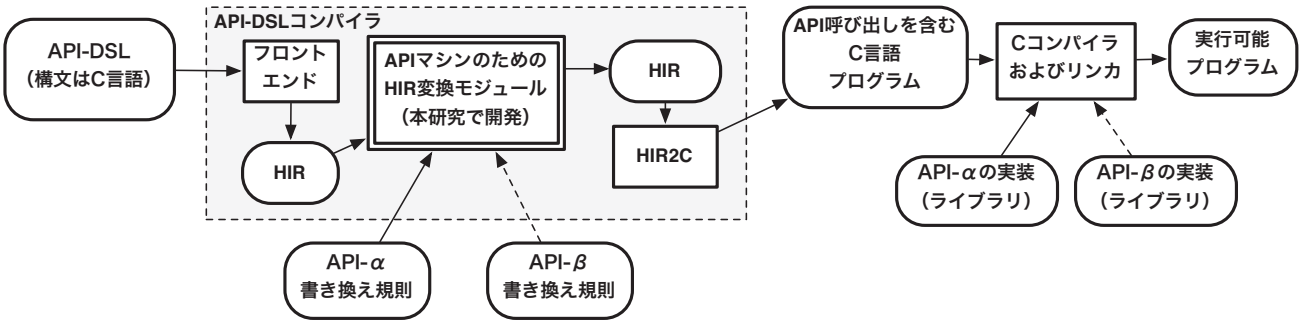

図 3 提案システムの全体像

```
 1: [System]
 2: RelatedNamePrefix = "_my_";
 3: OutputFunctionDefinition = NO;
 4:
 5: [Definition]
 6: type myType = "my_type_t";
 7: function myAddFunc = "my_add_func" : ( myType, myType ) -> myType;
 8:
 9: [Initialize]
10: addStringToTop( "#include <mytype.h>" );
11:
12: [SymTrans]
13: Var<int>[](){
14:   addSym( <myType>, relatedName( self.name ) );
15: }
16:
17: [HirTrans]
18: Var<int>[](){
19:   replaceExp( varFromString( <myType>, relatedName( self.name ) ) );
20: }
21:
22: AddExp<>[]( Exp<myType>, Exp<myType> ){
23:   replaceExp( functionStmt( myAddFunc, self.left.exp, self.right.exp ) );
24: }
25:
26: FunctionExp<>[ name("max") ](){
27:   replaceExp( functionStmt( myMaxFunc, self.param[0], self.param[1] ) );
28: }
29:
30: // 1行コメント
31: /*
32:    区間コメント
33: */
```

図 4 書き換え規則の記述例

必要とせずプログラムの書き換えのみを記述できることを目的として設計を行う．

提案するシステムにおける書き換え規則は，抽象構文木に対するパターンと操作で構成される．パターンは抽象構文木の部分木を表しており，入力プログラムの構文木に対して深さ優先探索の後順走査を行い，探索中のノードがパターンに一致すればそれに対応する操作を行う．操作はパターンに適合した抽象構文木の部分木に対して，置換，挿入，削除を行う命令列である．複数のパターンに一致する場合は，テキスト表現で先に記述されているパターンのみを適用する．置換や挿入により新しく追加された部分木に対してはパターンマッチを行わないので，書き換え規則を無限に適用し続けることはない．

3.3 書き換え規則の記法

書き換え規則の記述例を図 4 に示す．書き換え規則は記述する内容によっていくつかのセクションに分かれている．セクションのはじまりの宣言は，セクション名を各括弧([〜]) で囲み 1 行を占めることで表される．明示的にセクションの終わりを示すデリミタはなく，次のセクションのはじまりの宣言かファイルの終端でセクションは終わる．

コメントは C 言語と同様の構文を利用できる．すなわち，//から改行までと，/*と*/に囲まれた部分がコメントとして扱われる．

以降では各セクションにおける記述内容と記法について，図 4 に示す書き換え規則の記述例を用いて説明する．

3.3.1 System セクション

System セクションでは，本システムの動作に関連するパラメータの設定を記述する．パラメータの設定は name

```
1: Kind<Type>[Condition](ChildNodeList){
2:   Operation(ArgumentList);
3:   Operation(ArgumentList);
4:   ...
5: }
```

図 5 パターンと操作の凡例

= value; という形式で記述する．RelatedNamePrefix は relatedName 関数（後述）における前置詞となる文字列を表す．書き換え規則では，ダブルクォーテーションで囲まれた要素が文字列と認識される．OutputFunctionDefinition は書き換え規則で新たに定義された型や関数を，書き換え後のファイルに宣言として出力するかどうか指定できる．真偽値は，真の場合は YES，偽の場合は NO で表される．

この他にも書き換え規則を適用する関数を指定できる ApplicateFunctions や，uniqueName 関数（後述）の前置詞を指定する UniqueNamePrefix といったパラメータの設定が可能である．パラメータを明示的に指定していない場合は，システムのデフォルト値が設定される．

3.3.2 Definition セクション

Definition セクションでは，ユーザ定義の型や関数の宣言を記述する．型の宣言は type id = "TypeName";，関数の宣言は function id = "FunctionName" : (typeList) -> returnType; という形式で記述する．id は宣言の識別子であり，宣言した部分より後で参照することができる．TypeName や FunctionName はプログラムにおける関数名を指定する．typeList はカンマ区切りで引数の型を記述する．returnType は戻り値を指定する．

3.3.3 Initialize セクション

Initialize セクションでは，変換の前に行う操作を記述できる．パターンマッチの結果に関わらず行われるので，プログラム全体に一度だけ適用したい操作を記述する．図 4 の 10 行目の addStringToTop は，ファイルの先頭に文字列を挿入する操作であり，ライブラリ mytype のヘッダファイルをインクルードしている．置換などの部分木をターゲットとする操作は，Initialize セクションに記述できない．

3.3.4 SymTrans と HirTrans セクション

SymTrans セクションでは，シンボルテーブルに対するパターンと操作を記述する．HirTrans セクションでは，プログラムの抽象構文木に対するパターンと操作を記述する．SymTrans セクションには置換や削除などの操作に制限があるのみであり，パターンと操作の記法は HirTrans セクションと共通である．

パターンと操作の記法をを図 5 に示す凡例を用いて説明する．Kind はノードの種類を表しており，例として Var は変数，AddExp は加算演算子を表す．Type はノードの型を表しており，C 言語のプリミティブ型と Definition セクションで宣言したユーザ定義型を用いることができる．

```
int main(){
  double a, b, y;
  a = 77617.0;
  b = 33096.0;
  y = 333.75 * pow( b, 6.0 ) + pow( a, 2.0 ) *
    ( 11.0 * pow( a, 2.0 ) * pow( b, 2.0 ) -
    pow( b, 6.0 ) - 121.0 * pow( b, 4.0 ) - 2.0 ) +
    5.5 * pow( b, 8.0 ) + a / ( b * 2.0 );
  printf( "y = %.20e\n", y );
}
```

図 6 Rump's Example を C 言語で記述したプログラム

Condition は条件を指定することができ，複数指定する場合はカンマ区切りで記述する．パターンに適合する部分木に対して，中括弧（{}）内の操作列を上から順に実行する．操作の一覧を表 1 に示す．また，抽象構文木に対する直接的な作用はないが，部分木の生成に用いる関数を表 2 に示す．提案システムでは，これらの操作と関数を用いてプログラムの抽象構文木を書き換えることができる．

4. 適用事例：MPFR ライブラリ用 API-DSL コンパイラの構築

4.1 MPFR ライブラリの概要

MPFR[5] は GNU が開発した C 言語で利用できる任意精度浮動小数点演算ライブラリである．IEEE754[6] 規格由来の 4 つの丸めモードをサポートしている．

4.2 MPFR ライブラリのための API-DSL の設計

MPFR ライブラリを意識せずに C 言語でプログラミングできる環境を構築する．MPFR ライブラリのための API-DSL は，プログラムに現れる全ての倍精度浮動小数点数を任意精度浮動小数点数として扱う．すなわち，アプリケーション開発者は double 型変数を用いて C 言語でプログラミングするだけで，MPFR ライブラリを利用して高精度浮動小数点演算を行うアプリケーションを開発できる．MPFR ライブラリのため API-DSL コンパイラを構築する書き換え規則は，140 行で記述することができた．

4.3 評価

MPFR ライブラリのための API-DSL コンパイラの動作を確認するため，Rump's Example を用いた．Rump's Example は，浮動小数点演算の過程で深刻な桁落ちが発生するため，倍精度浮動小数点型では満足する結果を得ることができないワークロードである．Rump's Example を C 言語のプログラムで記述したものを図 6 に示す．図 6 のプログラムを MPFR ライブラリのための API-DSL 記述とみなして，提案システムを用いて変換を行ったところ正常に変換できることを確認した（変換後のプログラムは大規模なものとなってしまい，本稿では紙面の都合上掲載できなかった）．得られたプログラムを C コンパイラを用いて実行すると，浮動小数点数の精度向上によりアプリケーショ

表 1 操作一覧

操作名（引数リスト）	説明
replaceStmt(Stmt)	パターンに一致した部分木を，第一引数の Stmt で置換する．
replaceExp(Exp)	パターンに一致した部分木を，第一引数の Exp で置換する．
addStmtPrevious(Stmt)	パターンに一致した文の直前に，第一引数の Stmt を挿入する．
addStmtNext(Stmt)	パターンに一致した文の直後に，第一引数の Stmt を挿入する．
addStmtToBlockTop(BlockStmt, Stmt)	第一引数の BlockStmt で表されるブロックの先頭に第二引数の Stmt を挿入する．
addStmtToBlockBottom(BlockStmt, Stmt)	第一引数の BlockStmt で表されるブロックの末尾に第二引数の Stmt を挿入する．
deleteThisStmt()	パターンに一致した部分木を削除する．
addSym(Type, String)	第一引数の型であり第二引数の識別子である変数をシンボルテーブルに新たに宣言する．

表 2 関数一覧

戻り値の型　関数名（引数リスト）	説明
Stmt functionStmt(Function, ...)	第一引数の Function で参照される関数を作成して Stmt 型として返す． 第二引数以降は Function で表される関数の引数を Exp 型で指定する．
Exp functionExp(Function, ...)	functionStmt と同じであるが，戻り値が Exp 型となる．
Exp varFromString(Type, String)	識別子名が第二引数の String で型が第一引数の Type である変数の参照を Exp 型で返す．
String relatedName(String)	第一引数の String に RelatedNamePrefix を前置詞として接続した文字列を返す．
String uniqueName()	他の識別子と被らない一意の文字列を返す．

ン開発者の期待する結果を得ることができた．アプリケーション開発者は図 6 に示す C 言語のプログラムを記述するだけで良いため，提案システムを用いることで MPFR ライブラリの仕様や制約を意識することなく利用できる環境が構築できたといえる．

5. 評価と考察

5.1 提案するフレームワークの評価

前章で適用事例として挙げた MPFR ライブラリのための API-DSL コンパイラを，本稿で提案するフレームワークを用いずに開発する手間を考える．MPFR ライブラリのための API-DSL コンパイラを COINS のモジュールとして作成した試作版 [2] は Java のコードで 1136 行であった．提案するフレームワークを用いたとき，MPFR ライブラリのための API-DSL コンパイラを構築する書き換え規則は 140 行であった．書き換え規則の構文を習得する手間を考慮したとしても，記述量の差が顕著に出ており提案システムを用いることで効率的に拡張言語の開発が行えたといえる．

5.2 より高度な変換への対応

我々は，本研究室で開発中のベクトルコプロセッサ [7][8] のための開発環境の構築を進めている．このベクトルコプロセッサは，4 倍精度浮動小数点演算をハードウェアでベクトル実行することができる．ベクトルコプロセッサ向けのプログラムは，ベクトル演算に相当する API 呼び出しや，演算に用いるデータの授受を行う API 呼び出しの記述が必要である．これはアプリケーション開発者にとって手間のかかる作業であり，ベクトルコプロセッサを対象とした API-DSL の利用ができれば大幅な記述効率の向上に繋がる．

書き換え規則を用いてベクトルコプロセッサ向け API-DSL コンパイラを作成するにあたりいくつか課題がある．ベクトル化可能性判定そのものは COINS の並列化可能性判定を応用することで実現できるが，その書き換え方法に工夫が必要である．ベクトル化が可能であると判定されたループにおいて，そのループブロック内の演算をベクトル演算に相当する API 呼び出しに置き換える．すなわち，特定の条件を満たす範囲に対して書き換え規則を適用することが求められる．また，入れ子関係などの複雑な条件が記述できれば，多重ループの最内側だけベクトル化するといった変換が可能となる．

このような，より高度な書き換えを実現することができれば，C 言語の API 呼び出しとして命令が用意されているハードウェアや，より複雑な書き換えを要するライブラリ用の API-DSL コンパイラを構築することが可能となる．

6. 関連研究

ライブラリの API 呼び出しを自動的に挿入する研究は知りうる範囲には存在しなかった．ここでは，ソースコードの書き換えを支援する研究に対して考察を行う．

ユーザの定義するテンプレートに従ったプログラムの自動書き換えを行うツールに Proteus[9] がある．これはリファクタリング作業の効率化を目的としており，プログラムの意味を変更する利用方法は想定していない．そのため，API による拡張機能を使えるようにするといった用途には向かない．

ソースコードの書き換えを行うツールに DMS[10] がある．これはソースコード片から抽象構文木を構築し，変換前の部分木と変換後の部分木を用いて変換を行う．一対一

に対応した置き換えを前提としているため，新たな変数の宣言や書き換え部分とは別の場所に初期化／終了処理の追加はできない．

ユーザの定義するパターン変換系に従ってソースコードの編集を支援する環境に TEBA[11] がある．プリプロセスを実行する前のソースコードや，断片的なソースコードに対して適用できることが特徴である．プログラムの属性を考慮したパターンマッチによる置き換えで実現しており，プログラムの意味を変更するような置き換えにも柔軟に対応できる．その反面，プログラムに忠実な抽象構文木を生成していないため，スコープルールを考慮した初期化／終了処理の挿入や，変数の型などの意味的な部分へのパターンマッチを必要とする変換には向いているとはいえない．

コンパイル時にソースコードの書き換えを行うプログラミングパラダイムとして，アスペクト指向プログラミングがある．アスペクト指向プログラミングは，横断的関心事をモジュール化することでコードの散在を防ぐことができる．その実装方式は様々であるため，本稿では具体的なアスペクト指向プログラミング言語として AspectJ[12] を挙げて比較する．

AspectJ は Java 言語を基にした汎用的なアスペクト指向プログラミング言語であり，プログラム内の選択の対象となる部分（特定のメソッドの先頭や末尾など）や動作（特定のメソッド呼び出しなど）に対して，追加・変更を行う機能を持っている．提案システムは構文木のパターンに一致する部分に対して書き換え操作を施す機能を持っており，プログラムの指定部分を書き換えるという点では同じ考え方であるといえる．実際に AspectJ のサンプルプログラムとして紹介されている特定の条件を満たすメソッドに対する機能の追加は，本研究で提案するシステムで実現できる．記述容易性や表現可能なパターンに差異はあるものの，提案システムはアスペクト指向プログラミングと類似した使い方をすることも可能である．

7. おわりに

本稿では，書き換え規則に基づく API ベース拡張言語のためのフレームワークを設計し実装した．このフレームワークを用いることで，書き換え規則を記述するだけで API-DSL コンパイラを容易に開発できる．適用事例として，MPFR ライブラリのための API-DSL コンパイラを構築する書き換え規則を作成し，拡張言語のコンパイラが容易に開発できることを確認した．今後の課題は，5.2 節で述べたような拡張を行い，より高度な書き換えを実現することである．

参考文献

[1] Pavel Holoborodko, MPFR C++, http://www.holoborodko.com/pavel/mpfr/, 2013 年 9 月 22 日参照.

[2] 塩出拓也, 川端英之, 北村俊明：API を用いた言語拡張のための COINS を用いた処理系実装の試み, 信学技報, vol.112, no.373, SS2012-49, pp.19–24, January 2013.

[3] COINS 開発グループ , COINS コンパイラ・インフラストラクチャ, http://coins-compiler.sourceforge.jp/, 2013 年 9 月 22 日参照.

[4] Chris Lattner, The LLVM Compiler Infrastructure, http://llvm.org/, 2013 年 9 月 21 日参照.

[5] Fousse, L., Hanrot, G., Lefevre, V., Pelissier, P., and Zimmermann: MPFR: A multiple-precision binary floating-point library with correct rounding, ACM Transactions on Mathematical Software, vol. 33, no.2, Article 13, June 2007.

[6] Microprocessor Standards Committee of the IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008, August 2008.

[7] 金子啓太, 北村俊明：精度低下検出を行う浮動小数点演算器の検討と評価, 情報処理学会研究報告, vol.2011-ARC-193, no.16, pp.1–6, January 2011.

[8] Soseki Aniya, Toshiaki Kitamura: A Performance Improvement for Floating-Point Arithmetic Unit with Precision Degradation Detection, SASIMI2012, no.R4-13s, pp.490–491, Oita, Japan, March 2012.

[9] Collard, M.L. and Maletic: Document-Oriented Source Code Transformatin using XML, Proc. 1st International Workshop on Software Evolution and Transformation, pp.11–14, October 2004.

[10] Baxter, I.D., Pidgeon, C. and Mehlich, M.: DMS®: Program Transformations for Practical Scalable Software Evolution, Proc. 26th International Conference on Software Engineering, pp.625–634, IEEE Computer Society, Washington, DC, USA, May 2004.

[11] 吉田敦, 蜂巣吉成, 沢田篤史, 張漢明, 野呂昌満：属性付き字句系列に基づくソースコード書き換え支援環境, 情報処理学会論文誌, vol.53 no.7, pp.1832–1849, July 2012.

[12] 増原英彦：アスペクト指向プログラミング, コンピュータソフトウェア, vol.23 no.2, pp.4–28, April 2006.