Vol.2015-SE-187 No.24
2015/3/12

代数データ型によるC言語プログラムのモデル化

小嶋 一成[1,a)]　大久保 弘崇[1,b)]　粕谷　英人[1,c)]　山本 晋一郎[1,d)]

概要：代数データ型による C 言語プログラムのモデルを提案する．代数データ型によるモデルは CASE ツール記述の抽象度を高め，CASE ツール作成の負担を軽減する．C 言語の前処理系の構文は C 言語の文法とは独立しているため，前処理前のプログラムは代数データ型の木構造による表現と相性が悪い．C 言語開発者の視点で C 言語プログラムを捉えることを目的とし，提案モデルは前処理指令の構文に一部制約を課すことで，一貫性を維持したまま木構造に前処理前の情報を統合した．

1. はじめに

プログラム開発や保守では，多くの CASE ツールが使用される．特に下流工程で使用される CASE ツールでは，対象プログラムの字句解析や構文解析が必要となるものが多い．従来，それぞれの CASE ツールが個々に字句解析器や構文解析器を持っているが，それらの解析器を共有可能にし，CASE ツール開発の負担を減らせる CASE ツールプラットフォーム Sapid[1] が提案された．

Sapid では C 言語ソースの字句解析，構文解析，意味解析を行い，解析結果をソフトウェアデータベース (Software Database; SDB) として出力する．CASE ツール開発者はアクセスルーチン (Access Routines; AR) を使用して SDB にアクセスし，解析結果を利用する．開発者は字句・構文解析器を作ることなく CASE ツールを開発することができる．AR を使用した SDB へのアクセス処理は手続的であり，手順が多くなる傾向があり，記述の煩雑さと可読性の低さが課題であった．

我々は，字句解析や構文解析の解析結果を表現するモデルが解析結果へのアクセス処理のアルゴリズムに深く関わっている点に注目した．より簡潔な手順でアクセス処理を記述できるモデルとして，パターンマッチによる抽象度の高い記述を実現する代数データ型を選択し，代数データ型に基づく C 言語ソースモデルを提案する．

抽象構文木は代数データ型で同一の構造として表現できる．しかし C 言語において文脈自由文法で定義されるのは前処理後の構文である．一方前処理指令は C 言語の構文からは独立して記述できるため，前処理前の C プログラムを抽象構文木や代数データ型で表現することは困難である．我々は，前処理指令の構文に一部の制約を課すことで，前処理前の C プログラムを表現できる代数データ型を実現する．

2. 代数データ型

代数データ型は，複数の直積データ型の直和の形で表される．個々の直積データ型は，直和型として表される代数データ型のサブタイプであり，直和の種類を識別するタグとしての値コンストラクタと直積を構成するデータ型から成る．

代数データ型の値は，値コンストラクタを関数とみた関数適用の形式で直積を構成する値の列から構築できる．パターンマッチは直和の種類を指定した多方向分岐と直積データを構成する値の抽出を同時に記述できるため，関数型のプログラミング言語と相性が良く，簡潔な記述が可能となる．

多方向分岐は手続き型言語では switch 文，オブジェクト指向言語ではメソッドのオーバーライドより定義されるが，パターンマッチの方が記述量が少ない．

3. 既存の C 言語プログラムのモデル

C 言語プログラムのモデルは CASE ツール開発に必要となる C 言語プログラムの字句・構文情報を表現する．多くの場合、モデルは解析対象プログラムの構文を再現できることが求められる．また場合によっては、コメントやインデントなどのプログラムの動作と関係のない情報や，プログラムを意味解析した結果も求められる．

[1] 愛知県立大学 情報科学部
School of Information Science and Technology, Aichi Prefectural University
[a)] kojima@yamamoto.ist.aichi-pu.ac.jp
[b)] ohkubo@ist.aichi-pu.ac.jp
[c)] kasuya@ist.aichi-pu.ac.jp
[d)] yamamoto@ist.aichi-pu.ac.jp

3.1 Sapid による C 言語プログラムのモデル

Sapid では C 言語プログラムを I-model と P-model の 2 つのモデルで表現する．I-model は 13 種類のクラスと 29 種類のクラス間の関連で構成され，前処理後の C 言語本来の構文に関する情報を表現する．クラスにはそれぞれクラス固有の属性が付随している．クラス同士は特定の関連で相互に参照している．P-model は 10 種類のクラスと 15 種類のクラス間の関連で構成され，前処理に関する情報を表現する．C 言語プログラムを，マクロやディレクティブなどの前処理情報とそれ以外の部分で分割し，それぞれのブロックにクラスを割り当てた構造となっている．

Sapid によるモデルでは，複数のソースファイルからなるプログラムを，複数のファイル間で関連付けられた記号表を生成して解析することができる．クラスや関連の実体にはそれぞれ識別番号が割り振られる．ブロック内の同じ変数のように，同じ変数を示す実体には同じ識別番号が割り振られる．逆に，同名であるがスコープの異なる変数を表す実体には別の識別番号が割り振られる．このように Sapid では細かい情報を取得できる記号表が用意される．

3.2 Sapid によるモデルの実装

Sapid はモデルを扱う手段として AR を C 言語用ライブラリの形で提供している．AR を用いてモデルから特定の実体を取得するには，プログラム全体を示すクラスの実体から開始し，関連を辿ることによる実体の取得と，実体が持つ属性によるフィルタリングを交互に行う．

関数名を出力する小さな CASE ツール中にある，プログラム全体から関数を表す実体を取得する処理の例を図 1 に示す．1 行目はプログラム全体を表す実体の ID を取得している．2 行目はプログラムとファイルの関連から，ファイルの実体の ID を取得するイテレータの初期化をしている．3 行目はファイルの実体の ID をイテレータを使って順に取得し，それぞれの実体を処理している．4 行目はファイルと宣言の関連から，宣言の実体の ID を取得するイテレータの初期化をしている．5 行目は宣言の実体の ID をイテレータを使って順に取得し，それぞれの実体を処理している．6 行目は宣言の種類を示す属性を取得し，関数であるものをフィルタリングしている．7 行目は `printFunc`関数に関数の宣言を示す ID を渡し，関数名を出力している．10,12 行目は使用したメモリの開放である．

3.3 Language.C による C 言語プログラムのモデル

Language.C では C 言語プログラムを 34 種類の代数データ型で構成される抽象構文木 (Abstract Syntax Tree; AST) で表現する．C99 までの C 言語規格と GNU 拡張に対応しているため，Sapid よりも広い範囲の C プログラムを解析することが可能である．C11 の規格には未対応である．

このモデルには 3 つの問題点がある．

```
1  prog_id = spdGetAnObjId("program",
       NULL);
2  file_csr = spdGetRelObjInit(prog_id,
       "prog_file", "program_id");
3  while ((file_id = spdGetRelObj(file_csr
       )) != SPD_NON_ID) {
4    decl_csr = spdGetRelObjInit(file_id,
         "file_decl", "file_id");
5    while ((decl_id = spdGetRelObj(
         decl_csr)) != SPD_NON_ID) {
6      if (spdGetAttrValInt(decl_id,
           "sort") == DECL_FUNCDEF){
7        printFunc(decl_id);
8      }
9    }
10   spdFreeCursor(decl_csr);
11 }
12 spdFreeCursor(file_csr);
```

図 1 Sapid における実体取得処理の例

- 複数のファイルから構成されるプログラムを解析できない．
- 前処理後のプログラムのモデルである．
- 記号表がない．

C プログラムは通常多くのソースファイルから構成される．しかし Language.C では，1 つの C ソースファイルに対して 1 つの AST が生成されるため，複数のファイルを関連付けた解析ができない．

C プログラムにおいて `#ifdef`などの前処理はマルチプラットフォームな開発などに欠かせないが，Language.C ではそのような前処理前のプログラム構造を表現できない．AST で表現されるのは，前処理系の出力内容である．

Language.C の代数データ型 AST モデルでは記号表の段階の意味解析も行われないため，識別子の参照とその宣言を対応づけたり，複数の参照の同一性の判定といった基本的な分析もモデル利用者の責務となる．

3.4 Language.C によるモデルの実装

Language.C のモデルは，解析したプログラムと同様のふるまいをするソースを復元するための字句・構文構造のみ持つ．さらに，AST の各ノードはアノテーションとして任意の型の値を 1 つ保持できる．Language.C のパーサーはアノテーションのデフォルト値として NodeInfo 型を保持させる．NodeInfo は，ソースファイル名，行，列からなる位置情報と，ソースコードに出現した順に割り振られる一意な番号である．

Language.C のモデルは，AST を構成する代数データ型をパターンマッチで指定して利用する．関数名を出力する小さな CASE ツール中にある，AST から関数を表すノードを取得する処理の例を図 2 に示す．1 行目は抽象構文木の

```
1 func (CTranslationUnit extDecls _) =
    mapM selFunDef extDecls
2   where
3     selFunDef (CFDefExt funcDef) =
        printFunc funcDef
4     selFunDef _ = return ()
```

図 2 Language.C におけるパターンマッチによる記述の例

ルートからグローバル領域の宣言や定義のリストを取り出し，それぞれに selFunDef関数を適用している．3 行目はパターンマッチにより，構文木のノードが関数定義であるとき，その子ノードを関数名を出力する関数 printFuncに適用している．4 行目は関数定義のパターンマッチに失敗した時に，何もしないことを定義している．

Language.C には AST を解析するためのモジュールがあるが，現在開発中である．AST の部分木から宣言を解析し，型の情報を取得する関数を提供するが，関数内部の解析や定数畳み込みを行う関数は用意されない [5]．また，記号表作成のための補助関数は提供されるが，記号表そのものはモデル利用者が独自に作成する必要がある．

4. 提案モデル

我々は，Language.C によるモデルを拡張し，複数のソースファイルの解析及び前処理前のプログラムの表現に対応したモデルを提案する．提案モデルは 37 種類の代数データ型で構成されている．表 1 に提案モデルの構成要素を示す．Lanuage.C のモデルから追加や改良を行ったものには* を付してある．

4.1 複数ファイルへの対応

Language.C のモデルは単一のファイルのみを表現する．通常複数のファイルから構成される C プログラムに対する意味解析のための共通の基盤を提供するために，複数のファイルを関連付けて解析できるようにモデルを改良する．

提案モデル中で複数のファイルを表現している部分を図 3 に示す．CProgramは Language.C から追加した代数データ型で，プログラム全体を表す．第 1 引数はプログラム自体の名前をとる文字列で，第 2 引数はプログラムを構成するファイルを表す CFileのリストである．

CFileは Language.C の CTranslationUnitに相当する代数データ型である．TranslationUnit は前処理後のソースを指すが File は前処理前のソースを指す．CFileは CTranslationUnitと同様にグローバルスコープの宣言や関数定義を表す CExternalDeclarationのリストを持つ．また，ファイル名を Stringとして保持するように変更した．

表 1 提案モデルの構成要素

型名	意味
CProgram	* プログラム (複数ファイル処理のため追加)
CFile	* ファイル (CTranslationUnit から変更)
CExternalDeclaration	* トップレベルの宣言 (前処理情報のため拡張)
CDeclaration	宣言
CFunctionDef	関数定義
CStructTag	構造体と共用体を区別するタグ
CStructureUnion	構造体や共用体
CEnumeration	列挙型
CDeclarationSpecifier	記憶クラス指定子・型修飾子
CStorageSpecifier	記憶クラス指定子
CTypeSpecifier	* 型指定子 (long long 型のため拡張)
CTypeQualifier	型修飾子
CAttribute	関数の属性機能
CDeclarator	宣言子
CDerivedDeclarator	宣言子の派生
CArraySize	配列の大きさ
CInitializer	初期化子
CInitializerList	複数の初期化子をまとめたもの
CPartDesignator	初期化子で初期化を行う変数や配列のインデックス指定
CStatement	* 文 (前処理情報のため拡張, ラベル関係を改良)
CCompoundBlockItem	複文の構成要素
CAssemblyStatement	インラインアセンブラのステートメント
CAssemblyOperand	インラインアセンブラのオペランド
CExpresstion	* 式 (前処理情報のため拡張)
CAssignOp	代入を行う演算子
CBinaryOp	二項演算子
CUnaryOp	単項演算子
CBuiltinThing	コンパイラに組み込みの要素
CConstant	定数
CInteger	int 型の定数
CFloat	float 型の定数
CChar	char 型の定数
CString	文字列の定数
CStringLiteral	文字列リテラル
CIdent	* 識別子 (アノテーションのための型との結合度を下げるため改良)
CMacroCall	* マクロ呼び出し (前処理情報のため追加)
CDirective	* ディレクティブ (前処理情報のため追加)

4.2 前処理情報の表現

C 言語の前処理系は C 言語の文法とは独立している．前処理指令は C 言語の構文と無関係に任意の箇所に記述可能なため，C 言語の抽象構文木の構造に影響を及ぼさずに任

```
1 data CProgram a
2   = CProgram String [CFile a] a
3
4 data CFile a
5   = CFile String [CExternalDeclaration
      a] a
```

図 3 複数ファイルの表現

```
1 #ifdef DEF
2   for (i = 0; i < 10; ++i) {
3 #else
4   while (i >= 0) {
5 #endif
6     func(i);
7   }
```

図 4 除外した条件付きコンパイル指令の例

意の前処理指令の情報を共存させることは不可能である。そのため多くの解析器では前処理後の C 言語の構文に基づいた C 言語ソースのみを対象としている．

一方プログラム開発者は前処理後の C 言語ソースを目にすることは少ない．多くの場合前処理前の C 言語ソースを扱っている．我々は，開発者視点に近い形で C 言語ソースを扱えるように，前処理指令も前処理後の構文木も同時に表現可能な，前処理前のソースと前処理後のソースの両者を兼ね備えたモデルを提案する．

4.2.1 前処理指令の制限

C 言語の抽象構文木を可能な限り維持できるような前処理指令のサブセットのみを扱うことで，この問題を解決する．扱える前処理指令を以下のものに限定する．

- 宣言，文あるいは式として完結しているマクロ
- 中が宣言，文あるいは式として完結している条件付きコンパイルのためのディレクティブ
- 宣言，文あるいは式を記述できる位置にあるディレクティブ

マクロ本体は宣言，文あるいは式として完結している必要がある．マクロが宣言，式または関数呼び出しと互換であれば，AST 上ではマクロの呼び出しをそれらと類似したノードで表現することで，前処理後の AST の構造を保った AST にできる．一方，マクロ展開後の文字列が式として完結していない場合，前処理の前後で AST の構造に一貫性を保つことはできない．

上記の条件により除外されるプログラムの例を図 4 に示す．条件付きコンパイルで囲まれた領域が文として完結していないため，このようなプログラムについて前処理前と後で整合性のある AST は構成できない．

条件付きコンパイル以外の前処理指令についても，宣言，文あるいは式を記述できる場所に限るという制限を設けた．本来これらは任意の場所に記述可能であるが，これを木構造に反映させると構造が煩雑になり，モデルの利用が難しくなると判断した．

これらの制限を満たさない前処理指令の使用は C 言語の文法に対して不自然な表現になるものが多く，制限は妥当なものと我々は考える．

4.2.2 モデル中での定義

表 1 に示したうち，CMacroCallはマクロの呼び出しを表し，CDirectiveはディレクティブを表す，前処理指令を表現するノードのための型である．これらの定義を図 5 に示す．

CMacroCallは呼び出すマクロ名を識別子として表すCIdentを保持する．さらに関数形式マクロであれば実引数を保持する．マクロ展開後の AST も保持する．

CDirectiveはディレクティブやマクロ定義を表現する．モデルが扱う前処理情報は**#define**, **#undef**, **#if**, **#elif**, **#else**, **#endif**, **#ifdef**, **#ifndef**, **#pragma**, **#line**, **#error** の 11 種類のディレクティブと，コンパイルオプションとして定義されたマクロ，プリプロセッサが自動で定義したマクロである．#のみの行である NULL ディレクティブには対応していない．

CDrctIncludeは**#include**を表し，取り込むファイル名を保持する．

CDrctDefineFileは**#define**を表し，マクロ名，関数形式マクロであれば仮引数の名前，置き換え後の文字列を保持する．

CDrctDefineOptionと CDrctDefineInitはそれぞれコンパイルオプションとして定義されたマクロ，プリプロセッサが自動で定義したマクロを表し，CDrctDefineFileと同等の情報を保持する．

CDrctUndefは**#undef**を表し，定義を削除するマクロ名を保持する．

CDrctIfは**#if**が使われるブロックを表し，条件式と，**#if**と**#endif**で囲まれた領域の AST を保持する．**#elif**や**#else**節の AST も保持する．囲まれた領域の AST をどのように保持するかは後述する．

CDrctIfdef と CDrctIfndef はそれぞれ **#ifdef** と **#ifndef** を表し，条件となっているマクロ名，囲まれた領域の AST を保持する．**#elif**節は条件と囲まれた領域の AST のタプルのリストで保持し，**#else**節で囲まれた領域の AST を Maybe な値として保持する．

CDrctPragmaは**#pragma**を表し，後に続く改行までの文字列を保持する．

CDrctLineは**#line**を表し，行番号を表す式と，定義されていればファイル名を保持する．

CDrctErrorは**#error**を表し，出力するエラーメッセージを保持する．

トップレベルの宣言を表す CExternalDeclarationと，文を表す CStatementと，式を表す CExpressionの 3 種類

```
#define MAX(l,r) (((l)>(r))?(l):(r))
  max = MAX(left, right);
```

図 6 関数形式マクロを使用した C コード断片

```
CExpr
  (Just
    (CAssign
      CAssignOp
      (CVar (Ident "max" ()))
      (CMCallExpr
        (CMacroCall
          (Ident "MAX" ())
          (Just
            [CVar (Ident "left" ())
            ,CVar (Ident "right" ())])
          (CCond
            (CBinary
              CGrOp
              (CVar (Ident "left" ()))
              (CVar (Ident "right" ()))
              ())
            (Just
              (CVar (Ident "left" ())))
            (CVar (Ident "right" ()))
            ())
          ()))))
  ()
```

図 7 CMacroCall の例

```
#define FOO 0
#if FOO <= 0
  func1();
#else
  func2();
#endif
```

図 8 #if ディレクティブの例

の代数データ型に，マクロ呼出とディレクティブを記述できるようにした．

関数形式マクロを使用した文の例を図 6 に示す．提案モデルでは CMacroCallを使って図 7 のように構成される．なおアノテーションは空値 ()としている．

4.2.3 条件付きコンパイルの表現

条件付きコンパイルは C ソースプログラムの任意の部分を差し替える機能を持つ．すなわち，条件付きコンパイルで囲まれた C 言語ソースは AST の任意の構造となりうる．これを AST で表現するため，多相型により子ノードの型を指定できるようにした．

図 8 の例の場合, 提案モデルによる前処理情報は CDrctIf を使って図 9 のように構成される．

```
(CDrctIf
  [
    (
      CBinary
        CLeqOp
        (CMacroCall
          "FOO"
          Nothing
          (CConst
            (CIntConst
              (CInteger 0 DecRepr
                 FlagUnsigned)
              ()))
          ())
        (CConst (CIntConst 0 ()))
    ,
      [
        CExpr
          (Just
            CCall
              (CVar (CIdent "func1" ())
                    ())
              [])
          ()])
  ] :: [(CExpression (), [CStatement ()
    ] )]
  (Just
    [
      CExpr
        (Just
          CCall
            (CVar (CIdent "func2" ())
                  ())
            [])
        ()]) :: Maybe [CStatement ()]
  ()) :: CDirective [CStatement ()] ()
```

図 9 #if ディレクティブの例の表現

5. 解析器の実装

C 言語プログラムを読み込み提案モデルのデータ構造を出力する解析器を実装した．その方法には,

(1) 独自に字句・構文解析器を実装する
(2) Language.C の解析器を拡張する
(3) Sapid の SDB から AR を用いて情報を取得する

などが考えられる．今回は (3) を選択した。

Sapid の SDB へアクセスする標準のライブラリ AR は C 言語用のライブラリである．一方モデルは Haskell の代数データ型を使用している．2 つの言語を組み合わせるため, Haskell の FFI (Foreign Function Interface) を使用する．

FFI を通して AR を呼び出すためのモジュールを作成し, Haskell で扱い易い代数データ型とリストを主体としたデータ構造を提供する．また，一部の関数は FFI を使用しても

```
1  data CMacroCall block a
2    = CMacroCall (CIdent a) (Maybe [CExpression a]) block a
3
4  data CDirective block a
5    = CDrctInclude      String a
6    | CDrctDefineFile   (CIdent a) (Maybe [CIdent a]) (Maybe String) a
7    | CDrctDefineOption (CIdent a) (Maybe [CIdent a]) (Maybe String) a
8    | CDrctDefineInit   (CIdent a) (Maybe [CIdent a]) (Maybe String) a
9    | CDrctUndef        (CIdent a) a
10   | CDrctIf           [(CExpression a, block)] (Maybe block) a
11   | CDrctIfdef        (CIdent a) block [(CExpression a, block)] (Maybe block) a
12   | CDrctIfndef       (CIdent a) block [(CExpression a, block)] (Maybe block) a
13   | CDrctPragma       String a
14   | CDrctLine         (CExpression a) (Maybe String) a
15   | CDrctError        String a
16
17 data CExternalDeclaration a
18   = CMCallExt (CMacroCall (CExternalDeclaration a) a)
19   | CDrctExt  (CDirective (CExternalDeclaration a) a)
20     ...
21
22 data CStatement a
23   = CMCallStmt (CMacroCall [CStatement a] a)
24   | CDrctStmt  (CDirective [CStatement a] a)
25     ...
26
27 data CExpression a
28   = CMCallExpr (CMacroCall (CExpression a) a)
29   | CDrctExpr  (CDirective (CExpression a) a)
30     ...
```

図 5 モデル中での前処理情報の定義

直接呼び出せないため，ラッパー関数を C 言語で記述する必要があった．

5.1 AR 呼び出しモジュール

現在の実装では，AR 呼び出しモジュールを使用することで Sapid の機能を呼び出すことができる．提案モデルの機能だけ使いたい場合は AR 呼び出しモジュールを使用する必要はない．

5.2 モデル構成モジュール

AR 呼び出しモジュールにより SDB にアクセスし，このモジュールで提案モデルを構築する．

Sapid が提供する意味解析の情報，ファイル上での位置，前後のコメントをまとめたデータ型を作成した．この型の値をアノテーションとして AST の各ノードに付与できるようにした．

Sapid の P-model が持つ前処理前の情報は前処理の前後でのソース位置の対応であり，構文解析の結果は前処理後の情報として I-model にのみ存在する．このため，条件付きコンパイル指令によりコンパイルされなかった部分は構文解析を行わない．一方提案モデルでは，実際にはコンパイルされない部分に関しても構文木を保持する構造とした．この違いにより，現状の解析器の実装ではディレクティブに関するノードの内容が不完全である．

5.3 モデルの利用方法

Language.C によるモデルと同様に，AST を構成する代数データ型をパターンマッチで指定する．ただし提案モデルでは AST のルート部分に複数ファイルを扱うためのノードが追加されている点と，マクロやディレクティブが追加されていることに注意する．

関数名を出力する小さな CASE ツール中にある，関数を表すノードを取得する処理の例を図 10 に示す．

1 行目は AST のルートからファイルのリストを取り出し，それぞれのファイルに `selExtDecl`関数を適用している．3 行目はファイルを表すノードからグローバルスコープの宣言や関数定義のリストを取り出し，それぞれに `selFunDef`関数に適用している．4 行目は関数定義にパターンマッチさせ，関数名を出力する関数 `printFunc`に子ノードを適用している．5 行目はマクロ呼び出しにパター

```
1 func (CProgram _ files _) = mapM
    selExtDecl files
2   where
3     selExtDecl (CFile _ extDecls _) =
        mapM selFunDef extDecls
4     selFunDef (CFDefExt funcDef) =
        printFunc funcDef
5     selFunDef (CMCallExt mCall) =
        selInnerBlockMCall mCall
6     selFunDef (CDrctExt drct) =
        innerDrctBlockWith selFunDef
        drct
7     selFunDef _ = return ()
8     selInnerBlockMCall (CMacroCall _ _
        block _) = selFunDef block
9     innerDrctBlockWith f node = ...
```

図 10 提案モデルにおける解析方法の例

ンマッチさせ，マクロ展開後の部分木に selFunDef関数を再適用している．6 行目はディレクティブにパターンマッチさせ，ディレクティブ内部のブロックに selFunDef関数を再適用している．7 行目は関数定義のパターンマッチに失敗した時に，何もしないことを定義している．8 行目はマクロ呼び出しの内部の AST に関数があれば関数名を出力する関数を適用している．

6. 評価

提案モデルを用いることで CASE ツールが簡潔な記述で開発できることを示すため，小さな CASE ツールを作成し，その中で頻出する処理である構成要素を取得する処理のステップ数を比較する．また，Sapid，Language.C，提案モデルについて，それぞれモデル自体とその実装に関して，機能を比較する．

6.1 構成要素を取得する処理のステップ数による比較

関数名一覧を取得する小さな CASE ツールを作成し，関数を表す構成要素へのアクセス処理のステップ数を比較する．Language.C によるモデルと提案モデルでは比較部分のコードは同じになるので，Sapid によるモデルと提案モデルの 2 つを比較する，

提案モデルのアプリケーション記述言語が Haskell であるのに対して，Sapid は C 言語を基本としている．公平に比較するため，Sapid によるモデルのプログラムは，C 言語によるプログラムと同様の計算を AR 呼び出しモジュールを介して Haskell で記述したコードで考える．

Sapid における関数を表す実体へのアクセス処理を図 11 に示す．次のようなアルゴリズムで行われている．

(1) プログラム全体を表すノードを取得する．
(2) ファイルを全て取得する．
(3) それぞれのファイルから全ての宣言を取得する．

```
1  printFuncs_main ast = do
2    programId <- getAnObjId sdb "program"
3    fileIds <- getRelObjs programId
               "prog_file" "program_id"
4    declIds <- liftM concat $ forM
       fileIds (\fileId -> getRelObjs
       fileId "file_decl" "file_id")
5    funcs <- flip filterM declIds
       (\declId -> do
6      sort <- getDeclSort declId
7      case sort of
8        DeclFunction -> return True
9        otherwise -> return False)
10   mapM printFunc funcs
```

図 11 Sapid 記述例:関数実体の取得

```
1 funcWith f node@(CFunDef _ _ _ _ _) =
    f node
2 funcWith _ _ = return ()
3 printFuncs_main ast = everything (\a b
    -> a >> b)
4   ((const $ return ()) `extQ` (funcWith
      printFunc)) ast
```

図 12 提案モデル記述例:関数ノードの取得

(4) それぞれの宣言の中から関数であるもののみを残す．
(5) 関数である宣言を出力用の関数に渡す．

提案モデルおける関数を表すノードへのアクセス処理を図 12 に示す．Haskell の Generics[4] 拡張機能を用いることで，提案モデルの複数の代数データ型をまたぐ処理が次のようにして簡潔に記述できた．

(1) 関数ノードに対するパターンマッチに成功した時に処理をする関数を定義する．
(2) パターンマッチに成功した時に使用する関数と，パターンマッチに失敗した時に使用する関数と，関数の結果を畳み込む関数を指定し，データ構造全体に適用する．

Sapid によるモデルは 10 ステップ，提案モデルは 4 ステップとなった．Sapid によるモデルに比べ，提案モデルでは半分以下のステップ数で同様の処理を実現することができた．さらに，提案モデルは処理したい構成要素に対する処理のみ記述すればよいので，目的の処理に集中して CASE ツールを記述できるようになっている．

なお，Sapid によるモデルにおける実体を取得する処理を本来の C 言語で記述した場合のステップ数は 12 である．

6.2 機能比較

モデル間の機能を比較した結果を表 2 に示す．

- CASE ツール記述に使用する言語は Sapid のみ C で，Language.C と提案モデルでは Haskell である．
- Sapid によるモデルと，提案モデルの Sapid を利用し

表 2 モデルの機能比較

	Sapid		Language.C		提案モデル	
	モデル	実装	モデル	実装	モデル	実装
CASE ツール記述言語	-	C	Haskell		Haskell	
記号表による意味解析	○	○	○	×	○	○
複数ソースファイル	○	○	×	×	○	○
前処理前の情報	○	○	×	×	○	×
GCC 拡張・C99 規格	×	×	○	○	○	×
ステップ数	10		4		4	

た実装では，記号表による意味解析により，同じ実体を指す要素が識別できる．

- Language.C によるモデルと提案モデルでは，意味解析の情報はアノテーションとして付与する．
- 複数ソースファイルは，Language.C によるモデルのみ対応していない．
- 前処理前の情報は，Language.C によるモデルでは対応していない．提案モデルでは対応しているが，実装は不完全である．
- GCC 拡張と C99 規格は，Language.C によるモデルと，それを拡張した提案モデルで対応している．しかし Sapid によるモデルと，Sapid に基いて実装を行った提案モデルの実装では，これらの規格に対応していない．

7. おわりに

7.1 まとめ

本論文は，CASE ツール開発の負担を軽減することを目的とし，代数データ型に基づく C 言語ソースのモデルを提案した．

簡潔な処理で CASE ツールを記述できるようなモデルとして Language.C によるモデルが提案されているが，複数ファイルを同時に解析することができないという制約と，前処理の情報を解析できないという制約があった．これらの問題を解消し，複数のファイルを関連付けて解析でき，前処理情報も表現できるモデルを提案した．表現できる前処理情報に条件を設けることにより，前処理情報を AST に埋め込み、これにより，通常の字句・構文解析器よりも開発者の視点に近いモデルを提供できるようになった．

実装は Sapid に基づいて行った．Haskell から C のライブラリを使用できるように FFI を使い，Sapid のライブラリを Haskell で使用できるようなラッパーモジュールを作成した．このラッパーモジュールを用いてモデルを構成するモジュールを作成し，モデルの実装を行った．Language.C によるモデルでは意味解析ができなかったが，提案モデルの Sapid に基づく実装では記号表による意味解析が可能となった．

提案モデルでは Sapid によるモデルを扱うコードと比較して半分以下のステップ数で同じ処理を実装することが可能となった．簡潔な処理で CASE ツールを記述できるようになり，CASE ツール開発の負担が軽減された．

7.2 今後の課題

今後の課題として，モデルと実装の観点からそれぞれ 1 つずつ課題を挙げる．

モデルの観点では，マクロに関する扱いはさらに改良する余地がある．提案モデルにおいて，マクロ呼び出しを表すノード `CMacroCall`は，マクロ展開後の AST を持つ．一方マクロ定義を表すノード `CDrctDefineFile`は，マクロ本体に関して構文解析を行わず，その文字列そのものを `String`型で持つ．これは例えば，『マクロの本体式は全体を括弧で括られているか』という条件は，提案モデルの AST を辿ることでは診断できず，改めてこの文字列を構文解析する必要があることを意味する．他に，マクロ呼出の引数を省略した記述をモデルは表現できない．

実装の観点では，解析対象の C プログラムを読み込む構文解析器を独自に実現する必要がある．今回は Sapid によるモデルから情報を取り出したため，GCC 拡張や C99 に未対応，前処理により削除されコンパイルされないプログラム部分の構文木が生成されない，といった制約が生じた．ただし，Sapid の構文解析器における前処理情報の扱いは他に類のない細粒度かつ高精度なものである．これをさらに発展させる必要のある，提案モデルを完全に生成できる構文解析器の実現は困難な課題である．

謝辞 本研究は科研費 24300006 の助成を受けたものである．

参考文献

[1] 福安直樹, 山本晋一郎, 阿草清滋. 細粒度リポジトリに基づいた CASE ツール・プラットフォーム Sapid, 情報処理学会論文誌, Vol.39, No.6, pp.1990–1998, 1998.
[2] Language.C, `http://trac.sivity.net/language_c/`
[3] Sapid Home Page (in Japanese), `http://www.sapid.org/index-ja.html`
[4] Generics, `https://wiki.haskell.org/Generics`
[5] A closer look at Language.C, `http://zwizwa.be/-/meta/20120115-114428`