2015年9月25日

株式会社トランス・ニュー・テクノロジー

「Omni XcalableMPコンパイラの  
C/C++フロントエンドの開発」報告書

目次

[1 本報告書の構成 2](#_Toc430919264)

[2 納品ソフトウェアの利用方法 3](#_Toc430919265)

[2.1 gitリポジトリの構成 3](#_Toc430919266)

[2.2 ビルド手順 3](#_Toc430919267)

[2.2.1 Clang-3.6 のインストール 3](#_Toc430919268)

[2.2.2 CtoXcodeMLのビルド 5](#_Toc430919269)

[2.3 ビルドされたツールの利用方法 5](#_Toc430919270)

[3 作業報告 7](#_Toc430919271)

[3.1 納品ソフトウェアの内部実装に関する解説 7](#_Toc430919272)

[3.1.1 各ソースコードについて 7](#_Toc430919273)

[3.1.2 実装に用いているC++デザインパターンについて 8](#_Toc430919274)

[3.2 テスト用ソースコードによる評価 11](#_Toc430919275)

# 本報告書の構成

本報告書は、本案件に関する解説書を兼ねた文書であり、下記の内容からなる。

* 納品されたソフトウェア（CtoXcodeML）の利用方法について  
  下記の各パートからなる。
  + gitリポジトリの構成
  + ビルド方法
  + ビルドされたツールの利用方法
* 作業報告書一式  
  下記の各パートからなる。
  + 納品ソフトウェアの内部実装に関する解説
  + テスト用ソースコードによる評価

# 納品ソフトウェアの利用方法

本案件で開発されたソフトウェアは、omni-compiler.org上に設置されたgitレポジトリでバージョン管理されており、納品物はここに置かれている。

## gitリポジトリの構成

* docs/ ディレクトリ
  + docs/CtoXcodeMLreport-20150925.docx  
    本ドキュメントである。
* src/ ディレクトリ  
  ソースコード一式を含む。

## ビルド手順

まず、Clang-3.6をソースコードからビルドし、 /usr/local 以下にインストールしておく必要がある。そのあと、ソースコードをビルドする。

### Clang-3.6 のインストール

Clang-3.6のビルドには、同バージョンのLLVMコアとcompiler-rtが必要である。いずれもhttp://llvm.org/releases/download.html#3.6.2 からバージョン3.6.2が取得できる。また同じページの先頭のリンクでsubversionを用いたリポジトリも公開されている。しかしそれらを利用して取得するよりも、github のミラーを取得したほうが利便性が高い（git checkoutだけで各バージョンのブランチに切り替えることができる）ため、そちらを利用する方法でビルド手順を示す。

% cd 作業用ディレクトリ  
% git clone https://github.com/llvm-mirror/llvm  
% cd llvm  
% git checkout release\_36

% cd 作業用ディレクトリ/llvm/projects  
% git clone https://github.com/llvm-mirror/compiler-rt  
% cd compiler-rt  
% git checkout release\_36

% cd 作業用ディレクトリ/llvm/tools  
% git clone https://github.com/llvm-mirror/clang  
% cd clang  
% git checkout release\_36

% cd 作業用ディレクトリ  
% mkdir build  
 ※llvmのビルド用のディレクトリを準備する。名前は何でもよい。  
% cmake –G Ninja ../llvm  
 ビルド用のサブディレクトリ・ファイル群が生成される。  
　　また、この時点ですでにCコンパイラとしてclangを用いることができる  
　　のであればCC=clang CXX=clang++ という環境変数を設定してcmakeを  
　　おこなってもよい（gcc/g++を用いるよりも高速にビルドが完了する）。  
% ninja  
 ※（PATH環境変数で探せる位置に無いなら適宜フルパスで指定する）  
 ビルドが実行される。マルチコアの場合には自動的に並列ビルドとなる。  
 初回は4コア2スレッドのCPU（8並列）でも二時間ぐらいかかる。  
 明示的に並列度を指定する場合は-jオプションを用いる。ビルドの終盤  
　　でリンクをおこなう際にはメモリ不足になりやすいので、ビルドに失敗  
　　した場合には並列度を下げてやり直すと良い。また、前述のようにして  
　　clang/clang++を用いた方がメモリ消費も少なくて済むようである。  
　　一度ビルドが成功したあとは、再ビルドする際にcmakeから実行する  
　　必要はなく、ninjaコマンドのみを実行すればよい。

以上は通常のRelease版のビルド手順である。これ以外に、デバッグ版のビルドを下記のようにしておこなうことができる。このようにすると、LLVMの各種のコマンド(optなど)で –debug や –debug-only オプションが使えるようになり、コンパイラ内部のデバッグに役立つ情報が得られるようになる。

% cmake –G ninja –DCMAKE\_BUILD\_TYPE=Debug ../llvm  
% ninja  
 ※ビルドを実行（既に実行済みであれば、必要な部分だけリビルド）する。  
 Releaseビルドに比べてさらに多くのメモリを必要とするので、  
 メモリ不足で失敗するようなら-jオプションで並列度を下げて調整する。  
% ninja install  
 ※/usr/local以下にインストールされる。

CMakeのこの他のオプション指定については<http://llvm.org/docs/CMake.html> に書かれているので、そちらを参照されたい。

### CtoXcodeMLのビルド

gitレポジトリから取得した src/ ディレクトリ内で make を実行すればビルドが実行され、 CtoXcodeMLという名の実行ファイルが生成される。

make distcleanをすると、ビルド結果が全て消される。

Make cleanをすると、ビルド結果のうち、もっともビルドに時間のかかるXcodeMlRAV.oを残して、その他を消す。

他にもMakefile内にいくつかターゲットが記述されているが、開発途上段階でのテスト用に準備されたものが多く、動作は保障されない。

## ビルドされたツールの利用方法

例として、

clang -I some-include-dir/ -D user\_defined\_macro=value foo.c

としてコンパイルするような foo.c を対象にする場合、

./CtoXcodeML foo.c -- -I some-include-dir/ -D user\_defined\_macro=value

のように「-- の前に対象ファイル名、 -- のあとにその他のオプション」として指定する。その他のオプションが不要なときも「--」が必要である（この -- が無いと ./compile\_commands.json というファイルが用いられる。これについての詳細は<http://clang.llvm.org/docs/JSONCompilationDatabase.html>を参照のこと）。

以下は、 -- よりも左側に指定することのできるCtoXcodeML独自のオプションについて解説する。

./CtoXcodeML -help

でオプションの一覧が出る。

-file - emit 'file'

-lineno - emit 'lineno'

-column - emit 'column'

…これらはそれぞれ、出力されるXcodeMLの要素にソースコードのファイル名情報・行番号情報・桁位置情報を付加するオプションである。 C\_Frontと同一の結果を得たい場合には –file と –lineno を指定する必要がある。

-disable-typeTable - disable <typeTable>

-disable-symbols - disable <globalSymbols>, <symbols>

-disable-declarations - disable <globalDeclarations>, <declarations>

…これらはそれぞれ、 XcodeMLの一部の出力を抑制するためのオプションである。主にデバッグ用に用いる。

-trace-typeTable - emit traces on <typeTable>

-fulltrace-typeTable - emit full-traces on <typeTable>

-trace-symbols - emit traces on <globalSymbols>, <symbols>

-trace-declarations - emit traces on <globalDeclarations>, <declarations>

-trace-rav - trace Recursive AST Visitor

…これらはそれぞれ、 XcodeMLの出力中にコメントとしてデバッグ情報を含めるためのオプションである。

-typenamemap=<string> - a map file of typename substitution

…これはXcodeMLのtypeTableで扱われる型名を置換するためのオプションである。置換したい型名と置換後の型名をそれぞれ空白区切りで交互に並べたファイルを準備し、そのファイル名を上記の<string>のところに指定する。ただしファイルの末尾に空白や改行があると正しく読み込めなくなるので、ファイルの最終行は改行無しで終わらなければならない。

これ以外にもいくつかのオプションがあるが動作保障はされない。

# 作業報告

## 納品ソフトウェアの内部実装に関する解説

### 各ソースコードについて

* XcodeMlVisitorBase.cpp, XcodeMlVisitorBase.h  
  下記の各種の \*Visitor.cpp, \*Visitor.h の実装の基底クラスである  
  class XcodeMlVisitorBase を準備している部分。  
  意味的にはこのさらに上位に CRTPパターンで書かれたRecursiveASTvisitor クラスを基底に持つのだが、RecursiveASTvisitorクラスは大量のメソッドを持つため、本当に各種の\*Visitorの基底クラスとして実装するとコンパイル時間が何倍にもなるので、pimpl イディオム相当の class RAVBidirBridge をつかって、下記のXcodeMlRAV のほうに RecursiveASTvisitor の実装の部分を隠蔽している。（CRTPパターンとpimplイディオムについては後述）
* XcodeMlRAV.cpp, XcodeMlRAV.h  
  clang の libtooling ライブラリ内の RecursiveASTvisitor.h を利用したクラスを実装している部分。 RAVBidirBridge をつかってclass XcodeMlVisitorBase との間で双方向に橋渡しをしている。
* TypeTableVisitor.cpp, TypeTableVisitor.h  
  clang の AST から XcodeML の <typeTable> 部を生成する部分。  
  また、 class TypeTableInfo というデータ構造を作成し、clang AST の QualType で示された値 (型の種別情報) とXcodeML の の type 名 (文字列) との対応関係を管理する。
* SymbolTableVisitor.cpp, SymbolTableVisitor.h  
  clang の AST から XcodeML の <globalSymbols> 部および <symbols> 部を生成する部分。
* DeclarationsVisitor.cpp, DeclarationsVisitor.h  
  clang の AST から XcodeML の <globalDeclarations> 部および <declarations> 部を生成する部分。この部分がCtoXcodeMLの中でもっとも大きな部分を占める。
* CtoXcodeML.cpp  
  main 関数部分。与えられたコマンドラインから AST を構成し、上記各 Visitor にAST を渡す部分。

### 実装に用いているC++デザインパターンについて

これらのソースコードでは、C++のデザインパターンとして下記の二つのテクニックを用いている。

#### CRTP (Curiously Recurring Template Pattern)

親クラスとして  
　　template <class T> class 親クラス { … };  
のようなものを準備しておいて、利用者は  
　　class 利用者クラス : public 親クラス<利用者クラス> { … };  
のようにして用いる、というデザインパターン。

「奇妙に再帰的なテンプレート」パターンという名前のとおり、親クラスの型引数のところに子クラスの型をいれてそれを継承するという、一見するととても奇妙な形をしている。これは、 template を使わずに下記のように書いた場合と、やりたいことはだいたい同じだと考えてよい。

まず親クラスを  
　　class 親クラス {  
 virtual 仮想メソッド(…) {デフォルトの定義};  
 普通のメソッド(…) { 上記仮想メソッドを使って何か実装; };  
 };  
と定義しておいて、このクラスを継承して利用する側では  
　　class 利用者クラス : public 親クラス {  
 仮想メソッド(…) override { 何か自分用の中身を実装; };  
 };  
のようにする。これにより、親の「普通のメソッド」の動作の一部が利用者クラス側で変更可能となる。

これに対し、CRTPを用いた書き方は次のようになる。

まず親クラスを  
　　template <class D> class 親クラス {  
 置き変え可能なメソッド(…) {デフォルトの定義};  
 普通のメソッド(…) {  
 static\_cast<D\*>(this)->置き変え可能なメソッド(…);  
 という形で呼び出して実装;  
 };  
 };  
としておいて、利用者側で  
 class 利用者クラス : public 親クラス<利用者クラス> {  
 置き変え可能なメソッド(…) { なんか自分用の中身を実装; };  
 };  
とする。これにより、  
 利用者クラス x;  
 x.普通のメソッド(…);  
という形で「普通のメソッド」を呼び出すと、テンプレート展開の結果  
 static\_cast<利用者クラス\*>(this)->置き変え可能なメソッド(…);  
が中で使われることになるので、結果的に「利用者クラス」の側で定義した「置き変え可能なメソッド」が用いられることになる。

virtual を使った場合との違いは下記のようになる。

* virtual を使うと、仮想関数テーブルという「関数のポインタ一覧」がオプジェクトの一部となって埋め込まれているので、 virtual メソッドの呼び出しはメモリアクセスを何度かやって関数ポインタを取得する、というオーバーヘッドがかかる。  
  一方、 CRTP では「テンプレートの実体化」の時点では型引数が静的に定まっているので、メソッド呼び出しは C の関数呼び出しと同等の「静的呼び出し」になる (静的呼び出しはインライン展開できるので、中身が小さいメソッドでは virtual の場合に比べて飛躍的に高速)。
* virtual を使っている場合には  
   親クラス \*pointer = new 利用者クラス(…);  
  のような形で基底クラスのポインタによる抽象化が可能だが、CRTP の「親クラス」というのは実際には単一の基底クラスではない (型引数を入れるごとに違う型ができる) ので、このようなことはできない。つまり、実際には「親クラス」というよりは、単に「同じソースコード片となる部分を共通化して必要に応じて展開いるだけ」といった方が良い。

CRTP を一言でいうと「静的な情報を増やしてコンパイル時点でコンパイラが頑張ることで、利用者の自由度と実行時の高速さの両立をはかったもの」と言える。

#### Pimplイディオム (Pointer to implementation)

CRTP とはまったく逆に、コンパイル時点での静的情報をなるべく分離してソースコード変更の際のリコンパイルの量を減らそうというイディオム。

具体的には、ヘッダファイルに  
　　class 外に見せるクラス {  
 private:  
 class 内部的なクラス \*pimpl;  
 public:  
 コンストラクタ();  
 何らかのメソッド(…);

};  
のような形だけを公開し、 \*.cpp の方 で  
 class 内部的なクラス {  
 …実装すべて…;  
 };  
 外に見せるクラス:コンストラクタ() {  
 pimpl = new 内部的なクラス();  
 }  
 外に見せるクラス:何らかのメソッド(…) {  
 pimpl->何らかのメソッド(…);  
 }  
のようにして外のクラス経由で中の実装にアクセスできるようにする。

デザインパターンの Bridge パターンの応用の一種ともいえる。

## テスト用ソースコードによる評価