依存関係構築ツールmkdepとカーネルコンパイル ディレクトリ作成ツールconfigの汎用化 ～mipl関連研究～

奈良女子大学理学部情報科学科　05251864　山西康世

概 要

本研究はmakeを論理プログラミング言語Prologで実装する「Mipl」プロジェクトの関連研究である。makeにはMakefile作成のためのツール“ mkdep ”と“ config ”がある。それらのツールを汎用的に改良し、Mipl以外のmake代替プロジェクトでも使用できるようにした。本論文では、研究対象についての説明と改良点について述べる。

1　はじめに

miplとは、makeを論理型言語であるPrologで再実装するというプロジェクトである。先行研究として「Prolog Shell」があり、その研究では理論的にPrologでmakeの動作を実現することが可能であるとされている。しかし、過去には実用化に至ったものはなかった。そこで、本プロジェクトではその実証実験を行っており、最終的にはNetBSDのカーネルコンパイルが出来るようなレベルまで実装することを目標としている。

miplプロジェクトは2005年に立ち上げられた。現在、makeの動作実現のために必要とされる“ ターゲット構築ルールの検索 ”、“ 更新時間の比較 ”の実装、そして“ NetBSDのカーネル構築のためのmiplのデフォルトルール ”の作成がされている。現在までの研究ではNetBSDのシステムのうちcut等ソースの構造が単純なコマンドの構築に成功している。それを踏まえて本研究では、miplで実行されるMakefile自体をPrologで置き換えることを目指した。makeの実際の利用にあたってはその補助のためMakefileを作成するツールが使用されており、それらはmiplプロジェクトでも実装予定である。本研究ではMakefileを作成するためのツールである、“ mkdep ”と“ config ”の改良をテーマとすることにした。

mipl以外にもたくさんのmakeの代替プロジェクトがある。例えば、OmakeやGNUmake、Antなどがあり対象とするプログラミング言語もそれぞれ違う。そこで、改良にあたっては違う言語であっても汎用的に使用できるプログラムに変更することに重点を置くこととした。

2　mkdepとconfigの位置づけについて

前述の通り、本研究の対象はMakefile作成のためのツールmkdepとconfigである。mkdepは、Makefileに書いておかなければならないファイル同士の依存関係を自動で全て書き出してくれるプログラムである。そして、デバイスの認識や動作に必要であるデバイス接続に関する情報をシステムに与えるという「デバイスコンフィギュレーション」を行うプログラムがconfigである。どちらも、Makefile作成に大きく関わっているプログラムである。我々が進めてきたmiplプロジェク

トにおいても、他の make 代替プロジェクトにおいても、最終的に make したいプログラムが大規模であればあるほど、Makefile 作成のためのプログラムは必要である。mkdep と config を汎用的に使用できるプログラムに書き換えることは、make 代替プロジェクトに大いに役立つものだと言える。

以後、第 3 章で mkdep を、第 4 章では config をくわしく説明するとともに改良点や汎用性を述べる。

3 mkdep

3.1 mkdep について

大きなプロジェクトになればなるほど、Makefile に書かなければならない依存関係は莫大な量になる。そして書かなければならない依存関係も複雑になる。手作業で行うと変更が容易でないことに加え、漏れやミスがあるなど無駄な時間と労力がかかる。そのために NetBSD には mkdep というツールがある。mkdep とは、本来 Makefile に書いておかなければならない依存関係を、自動で生成してくれるプログラムである。

mkdep は C コンパイラへのフラグと C のソースファイルのリストを引数にとり、インクルードファイルの依存関係リストを構築する。そしてその結果を“ .depend ”ファイルに書き出す。オプションを使うことによってもっと make を便利に使うことが出来る。例えば、-a オプションを使うと結果を出力ファイルに追加することができる。それによって、同一の Makefile から、mkdep を複数回実行することが可能となる。また、出力先のファイルを指定したり、出力される依存関係の形式を変更したりすることもできる。

FreeBSD や OpenBSD にも同名で同機能のものが組み込まれている。他に同じような働きをするプログラムとしては、makedepend や gcc -MM などが挙げられる。

3.2 mkdep への改良

mkdep は 3.1 で説明した通り、多くのプログラムをコンパイルする際とても有用なツールである。しかし、make を使うことを前提として作成されているため、他のプログラミング言語を対象とした make 代替プロジェクトでは簡単に使用することができない。そこで mkdep のプログラムを改良することでその問題点を解消する。

まず、mkdep.c という元あったファイルにはほとんど手を加えないようにし、mkdep-plus.h という新たに作成したヘッダファイルを読み込ませることで、汎用化を実現している。他のユーザーが個別に変更を加える部分を別のファイルに分離し、分かりやすくすることに重点を置いた。新たに作成した関数などは mkdep-plus.c に記述し、それぞれの関数において変更する際に手を加える部分を明確にしている。

汎用的に使用できるように改良した点を、C 言語で出力されていたものを mipl 用に Prolog に変更する例で説明する。

改良前の mkdep での出力は、このような形式になっている。依存関係が“ make　 rule ”で書かれている。「：」の前に書かれているものがターゲットであり、後ろに書かれているのがターゲットを作成する為に必要なコンポーネントとなっている。

```
mkdep.o: mkdep.c /usr/include/sys/cdefs.h /usr/include/sys/mman.h /

/usr/include/sys/appleapiopts.h /usr/include/sys/_types.h /
                    ...
/usr/include/sys/select.h /usr/include/sys/_select.h findcc.h
```

それに対し、改良後の出力は、このように Prolog の述語の形で表されている。depend_file という述語の第一引数にターゲット、そして第二引数にコンポーネントのリストを記述している。

```
depend_file( mkdep.o,[mkdep.c,/usr/include/sys/cdefs.h,/usr/include/sys/mman.h /

,/usr/include/sys/appleapiopts.h,/usr/include/sys/_types.h /
                    ...
,/usr/include/sys/select.h,/usr/include/sys/_select.h,findcc.h /

 ,mkdep-plus.h
] ).
```

また、デフォルトで出力されるファイル名は“ .depend ”であったが、make 代替プロジェクトでは対象とするプログラミング言語の違いからデフォルトで出力されるファイル名のサフィックスも異なってくる。そこで、mkdep-plus.h で、このようにマクロ定義し変更が容易にした。ただし、-f オプションによるファイル名の指定は、改良後も使用できるようにプログラミングしている。

```
#define DEFAULT_FILENAME "depend.pl"
```

上記のように改良したプログラムを、元あった機能と共存させるためオプションを使用することによって出力をいつでも変更できるようにした。よって、元々あった-a オプションなどと同様に、-n オプションに私が作成した上記のような機能を持たせた。

この方法は mipl 以外の make 代替プロジェクトのために改良する場合にも適用できる。

4 config

4.1 config について

config とは、BSD 系 UNIX においてカーネルコンパイルディレクトリを造るためのツールである。具体的には、デバイスコンフィギュレーションを行いデバイスドライバを動作させるための情報を作成する。デバイスコンフィギュレーションは、デバイスの認識や動作に必要なバス名、そのバス上のアドレスなど、デバイス接続に関する情報をシステムに与える動作である。

config は、カーネルコンパイルディレクトリを作成する際にコンフィギュレーションファイルと files ファイルを読み込み、オプション依存の取り扱いの指定やカーネルを構成するソースファイルとオブジェクトファイルの選択の処理を行う。config ユーティリティは最初に機種名とアーキテクチャ名を読み取り、その結果によって読み込む files ファイルを選択する。それにより、config 自身はどのアーキテクチャ上で実行されているかや、機種に依存することが無い。

出力されるディレクトリの中身は、カーネルをコンパイルするための Makefile、ヘッダファイル、デバイス情報を格納したファイル、圧縮したデバイスコンフィギュレーションの情報を格納したファイルなどである。

本研究では config の Makefile 作成部分のソースの汎用化を対象とする。Makefile 作成部分ではまず、機種依存の部分が考慮された Makefile の元となるファイルを読み込み、そのファイルに書かれている通りに Makefile を作成してゆく。作成された Makefile は C コンパイラの利用を前提に書かれている。

4.2 config への改良

4.1 で説明した通り、config は確実にカーネルコンパイル用の Makefile を自動で作成してくれるというとても便利で効率の良いツールである。しかし、config によって作成されるカーネルコンパイルディレクトリの Makefile は C コンパイラの利用を前提に書かれており、他のプログラミング言語で行われている make 代替プロジェクトではそのまま使用することが出来ない。そこで config のプログラムを改良することでその問題点を解消する。

まず、mkdep と同様にして元あった mkmakefile.c というファイルにはほとんど手を付けないようにし、mkmakefile-plus.h という新たに作成したファイルを読み込ませることで汎用化を実現した。また、Makefile の元となるファイル Makefile.i386 をプログラミング言語の違いなどによって変更した MakefilePlus.i386 を新たに作成し、そのファイルを読み込めるように手を加えた。そして、使用プログラミング言語が違う make 代替プロジェクトで使用する際にも簡単に変更が出来るように新たに作成したファイルを工夫した。

mkdep と同様に、汎用的に使用できるように改良した点を C 言語で出力されていたものを mipl 用に Prolog に変更する例で説明する。

読み込む際に元々ある C 言語の Makefile 作成機能と共存させるため、改良の際追加した“ Prolog で Makefile を作成する ”という機能を mkdep 同様オプション機能として追加することによって汎用化を実現した。

Prolog で Makefile を作成するには、Makefile の元となるファイルも Prolog の述語の形式で書く必要がある。そこで、Prolog で書かれた Makefile の元となるファイルを新たに作成した。config を実行する際にオプションがついていれば、新たに作成した MakefilePlus.i386 というファイルを読み込む。Prolog 以外の言語に変更したい場合は MakefilePlus.i386 を変更すればよい。

```
<Makefile.i386>

...
MACHINE_ARCH= i386
USETOOLS?= no
NEED_OWN_INSTALL_TARGET?=no
.include <bsd.own.mk>
##
## (1) port identification
##
I386= $S/arch/i386
GENASSYM_CONF=
${I386}/i386/genassym.cf
```

```
##
## (2) compile settings
##
CPPFLAGS+= -Di386
AFLAGS+= -x assembler-with-cpp
-traditional-cpp
 …
```

```
<MakefilePlus.i386>

…
['MACHINE_ARCH','=',['i386']].
['USETOOLS','?=',['no']].
['NEED_OWN_INSTALL_TARGET','?=',['no']].
.include <bsd.own.mk>
##
## (1) port identification
##
['I386','=',['$S/arch/i386']].
['GENASSYM_CONF','=',['${I386}/i386/genassym.cf
']].
##
## (2) compile settings
##
['CPPFLAGS','+=',['-Di386']].
['AFLAGS','+=',['-x','assembler-with-
cpp','-traditional-cpp']].
…
```

また、config の処理により書き出されるファイルの属性の定義なども Prolog の述語の形式に変更するため、書き出す部分の関数を新たに作成した mkmakefile-plus.c に定義した。元々、config では書き出す処理によって関数を使い分けていたので、その部分をオプションがついている場合はすべて新しく定義された関数を使うようにプログラミングした。新しく関数を作成することで、汎用化を実現し変更を容易にしている。

実際の config のプログラムでは、関数を使い分ける際に下のようなループを使用していた。そこで、読み取る文字列とそれとセットになっている関数名の部分を構造体に格納し、make で実行する場合と make 代替プロジェクトで実行する場合で構造体の中身を変更できるようにした。

```
while (fgets(line, sizeof(line), ifp) != NULL) {
        lineno++;
        if (line[0] != '%') {
                fputs(line, ofp);
                continue;
        }
        if (strcmp(line, "%OBJS\n") == 0)
                fn = emitobjs;
        else if (strcmp(line, "%CFILES\n") == 0)
```

```
                fn = emitcfiles;
        else if (strcmp(line, "%SFILES\n") == 0)
                fn = emitsfiles;
        else if (strcmp(line, "%RULES\n") == 0)
                fn = emitrules;
        else if (strcmp(line, "%LOAD\n") == 0)
                fn = emitload;
        else if (strcmp(line, "%INCLUDES\n") == 0)
                fn = emitincludes;
        else if (strcmp(line, "%MAKEOPTIONSAPPEND\n") == 0)
                fn = emitappmkoptions;
        else {
                xerror(ifname, lineno,
                    "unknown %% construct ignored: %s", line);
                continue;
        }
        (*fn)(ofp);
}
```

変更後のループはこのようになっている。

```
while (fgets(line, sizeof(line), ifp) != NULL) {
        lineno++;
        if (line[0] != '%') {
                fputs(line, ofp);
                continue;
        }
        if (strcmp(line, name[0].macro) == 0)
                fn = name[0].function;
        else if (strcmp(line, name[1].macro) == 0)
                fn = name[1].function;
        else if (strcmp(line, name[2].macro) == 0)
                fn = name[2].function;
        else if (strcmp(line, name[3].macro) == 0)
                fn = name[3].function;
        else if (strcmp(line, name[4].macro) == 0)
                fn = name[4].function;
        else if (strcmp(line, name[5].macro) == 0)
                fn = name[5].function;
        else if (strcmp(line, name[6].macro) == 0)
                fn = name[6].function;
        else {
                xerror(ifname, lineno,
                    "unknown %% construct ignored: %s", line);
                continue;
        }
        (*fn)(ofp);
}
```

name という構造体の中身を書き換える関数を、mkmakefile-plus.c に定義しているので Mipl 以外の make 代替プロジェクトで使用する場合はその関数での書き換える内容を変更すれば良い。以上のようにして改良したプログラムは、mkdep と同様にオプションとして追加することにした。元

のオプションと同様にして、-l オプションをつけることで Prolog の述語形式で記述された Makefile を作成することができる。

この方法は mipl 以外の make 代替プロジェクトのために改良する場合にも適用できる。

5　本研究のまとめと今後の課題

mipl プロジェクトでは、NetBSD のカーネルコンパイルが最終目標である。本研究によって、Prolog の述語の形式で記述された Makefile を作成するツールの準備が整った。今後、NetBSD のカーネルコンパイルができるよう細部にわたって mkdep と config を改良すること、そして、Prolog で書かれた Makefile を実行できるようにコマンド等を整備していくことが課題である。

また、本研究のテーマである mkdep と config の汎用化を行った際に mkdep と config の詳しいドキュメントが必要であることが分かった。mipl に関連するそれぞれのツールに関して、ドキュメントを作成することで mipl 以外でも使用可能なツールの改良に役立てたい。そして、より一層改良を進めることにより mipl と共に mkdep や config のツールもオープンソースにしバージョン管理をきちんと行える体勢を整えることも課題である。

6　謝辞

本研究にあたって、いつも何から何までご指導いただいている鴨浩靖先生、make の学習やアドバイスをしていただいてお世話になった新出尚之先生、そして mipl の事について普段からお世話になっている奈良女子大学大学院　野口真理子さん、 また論文へのアドバイスをいただいた奈良女子大学大学院　藤田恵さんに大いなる感謝の辞を申しあげます。

参考文献

[1] Masami Hagiya：Prolog Shell —Prolog with Modality IPSJ SIGNotes SYMbol manipulation Abstract No.026-002, 1983

[2] Andrew Oram,Steve Talbott 共著，矢吹道郎 監訳，菊池彰 訳: make 改訂版,O'REILLY,1997

[3] The NetBSD Project，`http://www.jp.netbsd.org/ja/`