計算機アーキテクチャ 134－15
（1999．8．3）

異なるコンパイラ中間表現の相互変換に関する考察

曽山典子 * 神戸和子 * 城 和貴 † 加古富志雄 †

* 奈良女子大学人間文化研究科
† 奈良女子大学理学部

概要

中間表現の研究は永い年月をかけて多くの研究者によって研究されてきた．中間表現の基本構成は共通の認識によって構築されているにもかかわらず，多くのコンパイラでは独自の中間表現を構築し，その実装に多くの時間を費やしている．近年，コンパイラ技術の融合を図るべく統一的中間表現が提案されているが，すでに開発されたコンパイラ技術との融合は困難である．本稿では，すでに実装された異なる中間表現を比較し，それらの相互変換の可能性を調べ，その結果，2つの異なる中間表現は実装上の細部では困難な部分もあるが，概ね可能であることがわかった．同時に各構造体の構成成分がおおよそ同じであることがわかった．

Converting Different Intermediate Representations of Parallelizing Compilers: A Case Study

Noriko Soyama * Kazuko Kambe * Kazuki Joe † Fujio Kako †

* Graduate School of Human Culture, Nara Women's University
† Faculty of Science, Nara Women's University

Abstract

Many researchers of parallelizing compilers have proposed various design and implementation schemes of intermediate representations (IRs) for the last several decades. The foundation of IRs seems to reach some common ideas, nevertheless they have actually spent much time to complete the foundation of IRs individually. Recently, the concept of universal IRs (UIRs) has been proposed for such general purpose schemes. According to the schemes, different IRs should be integrated even after their implementations, but it may be hard to obtain common UIRs practically. In this paper, we compare different IRs of two parallelizing compilers, then we show the possibility of the conversion of different IRs. As a result, we find that it is practically possible to convert different IRs except some implementation details of the IR data structures, and there are many similarities between different IRs.

1 はじめに

中間表現は並列化コンパイラ研究の基盤であり，これまでに数多く提案・設計されてきた．しかし，各コンパイラごとに並列化・最適化手法によって個々の表現を用いており，汎用的に使うことができる並列計算機技術を確立することは困難である．近年，コンパイラ手法の統合を図るべく，統一的中間表現（Universal Intermediate Representations :UIRs）が提案されている．[1, 3]

PROMIS(UIUC,UCI) は，高レベル並列性と低レベル並列性を一つの中間表現で記述できる UIR を実装レベルで実現しており，Narafrase（奈良女子大，奈良先端大，和歌山大）は，UIR の考えに基づいて，データ分割グラフ (Data Partitioning Graph: DPG)[7] を持った中間表現を実装し，分散メモリ環境用の自動並列コンパイラを開発している．

PROMIS と Narafrase は，相互の UIR およびそのデータ構造，コンパイラの並列化・最適化施行時の UIR にアクセスする方法等の定義を統一し，双方の統一的中間表現に結果として互換性を持たせるように現在設計，開発されているところである．しかし，この 2 つのシステムのように開発段階で連係している場合は相互の技術が容易に融合され得るが，既に開発されたコンパイラ技術との融合は困難である．

本稿では，既に開発された異なるコンパイラの中間表現を比較し，相互の中間表現への変換の可能性を示す．本論文の構成は以下の通りである．第 2 節では関連研究について述べる．第 3 節では異なる 2 つの中間表現 (MC と Narafrase) について述べ，比較を行う．第 4 節では異なる中間表現の変換の可能性について考察を行い，第 5 節でまとめを述べる．

2 背景

本節では，中間表現とコンパイラの実装についての関連研究を紹介する．一般的な中間表現には，木構造のグラフで式を表すために使われる AST(Abstract Syntax Tree) や，ループや条件分岐制御依存のような制御フローを表す有向グラフ CFG(Control Flow Graph) 等がある．種々の最適化手法で用いられる中間表現としては，タスク間の制御依存関係を表す有向グラフ CDG(Control Dependence Graph) や，タスク間のデータ依存関係を表す有向グラフ DDG(Data Dependence Graph) がある．

統一された中間表現を利用したコンパイラの実装としては，PROMIS や SUIF(Stanford University Intermediate Format)，OSCAR(Optimally Scheduled Advanced Multiprocessor, 早稲田大) 等がある．

SUIF は米国の NCI(National Compilar Infrastructure) の元となる研究プロジェクトの一つである．疑似機械命令の列のみでできている Low-SUIF と，条件分岐やループ，配列参照などを AST 形式で表した High-SUIF からなる統一された中間表現を利用している [3].

OSCAR はプログラムをマクロタスク（ブロック）に分けて，マクロタスク相互のデータ依存や制御依存を効率的に解析することができるマクロフローグラフ (Macro Flow Graph:MFG) とマクロタスクグラフ (Macro Task Graph:MTG) を中間表現として利用している [5].

Promis の中間表現は，階層的タスクグラフ (Hierarchical Task Graph:HTG)[4] を基本としている．フロントエンドとバックエンド双方から利用することが可能であり，並列プログラムの持つ，疎粒度・中粒度・細粒度の並列性を効果的に表現している [2].

3 異なる中間表現の比較

2 節でも述べた通り並列コンパイラには様々な中間表現が存在するが，これらは各コンパイラ開発者が個々の並列化・最適化手法において利用しやすい表現として構築されたもので，他のコンパイラの中間表現との互換性はない．PROMIS と Narafrase のように開発段階で連係している場合は相互の技術が容易に融合され得るが，既に開発された異なるコンパイラ技術との融合は困難である．しかし，異なるコンパイラ中間表現を統一的中間表現に変換する機能を相互のコンパイラに追加することによって，相互の技術利用が可能になる．

そこで，Narafrase とそれとは異なる中間表現で既に実装されている並列コンパイラ MC(Meta Computer Systems, 早稲田大) を例にあげ，2 つの中間表現を比較し，相互の変換の可能性を調べる．本節では，Narafrase と MC について述べ，2 つの中間表現を比較する．

3.1 Narafrase と MC

Narafrase は共有メモリ型から分散メモリ型に至るまでを対象とし，データの分割配置と通信情報を，統一した中間表現で表した自動並列化コンパイラである．Narafrase の中間表現は HTG を拡張したデータ分割グラフ (DPG) を使用している．DPG はプログラム中のすべての変数に対する各アクセスの存在と，その種類や依存関係を表現することができる．

MC は分散環境を対象とした並列コンパイラ及び実行環境で，中間表現は AST を基本構造とし，階層的グラフ概念を適用している [6].

どちらの中間表現も AST を基本とし，制御フロー，制御依存，データ依存に関する情報を持ち，階層的グラフ概念を適用しているが，データ構造や情報の持ち方はかなり異なっている．その上，MC は Java 言語で開発されており，C++で開発されている Narafrase のクラスを利用することも不可能である．

3.2 中間表現の比較

双方の中間表現をフレームワーク（ノード，アーク，シンボル，階層構造）について比較する．

ノード

Narafrase の中間表現は次のノードで構成されている．

Start　HTG の入口を表す
Stop　HTG の出口を表す
Expression　代入文
Return　関数からの復帰文 (Return 文)
GOTO　GOTO 分岐文
NOP　非実行文を表す
Toplevel　全ての HTG ノードのルートを表し，手続きに対応する
Floop　ループの繰り返し数が整数定数のループ文を表す
Gloop　*Floop* 以外のループ文を表す
BlockIF　IF 文を含むブロックを表す
Block　*BlockIF* ノード以外のブロックを表す

MC の中間表現は次のノードで構成されている．(==ノード) は，各ノードが相当する Narafrase のノードを示している．

Program　入力されたプログラムそのものを表す．全体の頂点となるノード (== TopLevel)
Main　プログラムのメインルーチンノード (== TopLevel)
Function　関数，サブルーチンを定義するための中間表現のルートノード (== TopLevel)
Doloop　FOR 型のループ制御文により実行されるループを表す (== Floop, Gloop)
Genloop　FOR 型以外のループ制御文を表す (== Floop,Gloop)
Macrobk　任意の連結グラフにより構成される．グラフをカプセル化したい時に使う (== Toplevel)
Basicbk　プログラム中の基本ブロック部分を表現するために用いる (== Block)
Branch　IF 文や SWITCH 文などの条件分岐を表す (== BlockIF,Block)
Merge　分岐ノードにより分割された制御の流れが合流する点に挿入される (== NOP)
Assign　代入文 (== Expression)
Call　サブルーチンコールを表す (== TopLevel)
Entry　制御フローグラフ始端点を表す (== Start)
Exit　制御フローグラフ終端点を表す (== Stop)

対応するノードを比較すると，Narafrase で *Toplevel* として生成しているノードを，MC では *Program, Main, Function, Call* として個々にノードを用意していることがわかる．これは，Narafrase の中間表現ではコンパイルモジュール単位で *Toplevel* ノードが生成され，プログラムの中での各 *Toplevel* ノードの構成，つまりプログラム中のプロシージャ間の呼び出し関係を Callgraph で知ることができるためである．

双方の中間表現のノードを比較して，かなりの相似点を見つけることができたが，制御文に関してはかなり相違点がある．MC では，IF 文や Switch 文などの条件分岐を *Branch* ノードで表し，*Branch* ノードから分かれた制御の流れが合流する点に *Merge* ノードを生成する．しかし，Narafrase では IF 文の論理式を *BlockIF* ノードで表し，実行文を *Block* ノード，またはその他の相当するノードに分けて生成し，MC の *Merge* ノードのような特別なノードは用意していない．*BlockIF* ノードは *Start,Stop* ノードを有する下位層を持ち，論理式は *Expression* ノードとして生成される．ループの処理に関しては，Narafrase では繰り返し数が整数定数であれば *FLoop* ノード，整数定数でなければ *GLoop* ノードを生成しているが，MC では FOR 型のループ制御文とそれ以外（WHILE,UNTIL 制御文及びフォワードアークを形成する GOTO 文で実行されるループ）に分け，繰り返し数は各ノードの構造体に持っている．

アーク

双方とも制御フローグラフ (CFG)，制御依存グラフ (CDG)，データ依存グラフ (DDG) をアークとノードを用いて表現しているが，リンク方法は異なっている．また Narafrase ではプログラムの変数ならびにタスクによる変数のアクセスを明示的に表現する DPG を利用しているが，変換する際には MC において同等のアークとノードを生成しなければならない．

シンボル　双方とも AST を基本として中間表現を構成している．

階層構造　双方とも階層的グラフ概念を中間表現に適用している．

4 相互変換の可能性

表1は中間表現のフレームワーク（ノード，アーク，シンボル，階層構造）について相互変換の可能性をその難易度でクラス分けし，変換処理の概要を示したものである．但し，階層構造についてはノードとアークの各変換処理に含まれるので，この表では独立した構成成分としては挙げていない．また各アークの変換については，アークの構造体だけでなくノードの構造体もまた同時に処理しなければならないが，この表では構成成分ごとに示しているので処理内容については重複している箇所もある．

表に示した通り，2つの中間表現の核となる構成成分は概ね同じであることがわかる．なぜならどちらの中間表現もASTを基本とし，制御フロー，制御依存，データ依存に関する情報を持ち，階層的グラフ概念を中間表現に適用しているからである．これら共通の構成成分はMCとNarafrase以外のコンパイラでも同様に必要としている構成成分であろう．そしてそれ以外の構成成分は，個々の中間表現によって異なる，並列化・最適化手法を行うために必要となる情報である．既に開発されたコンパイラの中間表現を変換する上で，AST，制御フロー，制御依存，データ依存に関する情報や階層的グラフ概念などを核の構成成分とし，それ以外の並列化・最適化施行時に必要となる情報を拡張構成成分として考え，統一的中間表現を構築することが課題と言えよう．

5 まとめ

異なるコンパイラNarafraseとMCの中間表現を比較し，相互の中間表現の変換は実装上の細部では困難な部分もあるが概ね可能であることが分かった．中間表現の変換が可能であることは，既に開発された並列コンパイラ技術を融合する可能性につながる．本考察で取りあげた2つの中間表現の共通する構成成分は他の並列コンパイラの中間表現においても含有される情報である．これを統一的中間表現の核となる構成成分として，それ以外の並列化・最適化を行うための変換処理の施行時に必要となる情報に関しては拡張構成成分として構築することによって，より汎用的な統一的中間表現を提案できると考える．今後は並列技術の融合を目標とする上で，既に開発されたコンパイラの中間表現との変換も考慮し，より汎用性のある統一的中間表現を提案していきたい．

謝辞　本研究の一部は日本学術振興会未来開拓学術研究推進事業（知能情報・高度情報処理分野）JSPS-RFTF96P00505の援助による．

参考文献

[1] C.Brownhill, A.Nicolau, S.Novack, C.Polychronopoulos. *Achieving Multi-level Parallelization.* LNCS Vol.1336:183–194, 1997

[2] H.Saito, N.Stavrakos, S.Caroll, C. Polychronopoulos and N.Nicolau. *The Design of the PROMIS Compiler,* Technical Report, UIUCC-SRD 1539(revised), March 1999.

[3] Robert Wilson et al. SUIF. *An infrastructure for research on parallelizing and optimizing compilers.* Technical Report, Computer System Laboratory, Stanford University.

[4] M.Girkar and C.D.Polychronopoulos. *The Hierarchical task graph as a universal intermediate representation.* International Journal of Parallel Programming, 22(5):519-551, Oct 1994.

[5] H.Kasahara, H.Honda, A. Mogi, A.Ogura, K.Fujiwara, and S.Narita. *A Multi-Grain Parallelizing Compilation Scheme for OSCAR(Optimally Scheduled Advanced Multiprocessor).* In Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing, 283–297, 1992

[6] Youichi Muraoka. *Parallelizing and Distributing Compilers for Meta-Computer Systems.* Technical Report, University of Waseda, 1997.

[7] Tsuneo Nakanishi, Kazuki Joe, Hideki Saito, Constantine D.Polychronopoulos, Akira Fukuda, and Keijiro Araki. *The Data Partitioning Graph: Extending Data and Control Dependencies for Data Partitioning.* Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing:170–185, Aug 1994.

表 1: 相互変換の可能性

Level	Component	Contents of Conversion
0	Symbol	双方とも AST を基本としているため，容易に変換できる．
1	CFEs(Control Flow Edges)	MC ではノード構造体の中にリンク情報を持ち，アーク構造体は用意していない．ノード構造体の中に階層構造のネストレベルを整数値で保持し，下位ノードのアドレス情報は配列で持っている．Narafrase ではノード構造体の中に階層情報の構造体を持ち，下位ノードのアドレス情報はリストで持っている．また制御フロー情報は CFE 構造体で持っている． ($Narafrase \longrightarrow MC$) Narafrase のノード構造体の中の階層情報から下位ノードの情報を取り出し，MC のノード構造体のネストレベルをカウントして求める．MC の制御フロー情報は CFE 構造体から取り出して変換する． ($MC \longrightarrow Narafrase$) MC のネストレベルから Narafrase の階層構造のリストリンク順を知ることができる．制御フロー情報は各ノードの下位ノード情報から取り出し，CFE 構造体を生成する．
2	CDEs(Control Dependence Edges)	一つのノードに複数の依存アークが集まる場合において，飛び先の情報の表し方が異なる．MC では OR 演算子で各アークを結んだ式を制御依存条件としている．Narafrase では当該ノードと飛び先ノードをラベルとして持っている． ($Narafrase \longrightarrow MC$)CDE のラベルから当該ノードと飛び先ノードの情報を取り出し，MC の制御依存条件を生成する． ($MC \longrightarrow Narafrase$) MC の制御依存条件から Narafrase のラベルを生成する．
3	DDEs(Data Dependence Edges)	MC ではノード構造体に定義／参照の配列があり，下位ノードにおいて定義／参照されている変数が上位ノードの配列に格納されている．これにより階層的なデータ依存関係を表現している．Narafrase では同一階層での依存関係を表す構造体と全階層にわたる依存関係を表す構造体の 2 つを用意している．また MC には依存タイプ（フロー依存，逆依存，出力依存）の情報があり，Narafrase にはデータアクセスコスト情報 (当該変数のサイズと当該タスクが変数に Read アクセスする回数の積で求まる) がある． ($Narafrase \longrightarrow MC$)DPG のデータアクセス枝と CDE の情報で依存タイプを判断する．同時に同一階層の依存関係構造体と全階層の依存関係構造体の情報からノード構造体の定義／参照の配列を生成する． ($MC \longrightarrow Narafrase$) ノード構造体の定義／参照の配列からデータアクセスコストを求める．同時に同一階層の依存関係構造体と全階層の依存関係構造体を生成する．
4	DPG	($MC \longrightarrow Narafrase$) DPG は Cnode 集合 (タスクノード集合),Dnode 集合 (タスクによってアクセスされる変数集合),RAEs(Read Access 枝集合)WAEs(Write Access 枝集合) からなり，アルゴリズム [7] を使って生成する．CDG と DDG の情報 (Node 集合,$\{DDEs, CDEs\}$) を入力し，(Cnode 集合, Dnode 集合,CDE 集合,DDE 集合,RAE 集合,WAE 集合) を出力する．

5	Callgraph	Narafraseでは Callgraph で関数手続きの関係を表している．MC では *Program* ノードを先頭ノードとし，*Main* ノードがその直下に置かれ，その下位層にその他のノードが続く．関数・サブルーチンは *Function* ノードとして生成し，関数・サブルーチンを定義するためのルートノードとなる．このノードには直接，枝を張られることはなく，*Call* ノードで呼び出されるシンボル (関数シンボル) を設定して関係づけている．メインルーチン，関数・サブルーチンにはユニークなプログラム単位番号が設定されており，これによりプログラム全体の流れがわかる． ($Narafrase \longrightarrow MC$) Callgraph 構造体の関数の呼び出し関係情報を見て，MC の *Program,Main,Function,Call* 各ノードを生成する．同時に各アークのリンク情報も付加する． ($MC \longrightarrow Narafrase$) MC の *Program,Main,Function,Call* 各ノードの呼び出し関係を Callgraph 構造体に変換する．
6	Node	3節で示した通り，互いに対応するノードは各データ構造に合わせて変換することが可能である．制御文 (IF 文と Loop 文) については，以下のように変換する． $Narafrase(BlockIF, Block) \Leftrightarrow MC(Branch, Merge)$ ($Narafrase \longrightarrow MC$) *BlockIF* ノードの下位層の *Expression* ノードにある分岐条件式を使って，*Branch* ノードを生成する．*NOP* ノード (*BlockIF* ノードから分岐した *Block* ノードが合流するノード) のリンク情報から *Merge* ノードを生成する． ($MC \longrightarrow Narafrase$) *Branch* ノードの情報から，*BlockIF* ノードを生成し，その下位層に *Start,Stop* ノードとともにノード分岐条件式を *Expression* ノードとして生成する．*BlockIF* ノードと *Block* ノードが合流する点として *NOP* ノードを生成する． $Narafrase(Floop, Gloop) \Leftrightarrow MC(Doloop, Genloop)$ Narafrase ではループ条件文が *Floop* または *Gloop* ノードの下階層にある *Expression* ノード (*BlockIF* ノードの下位層にある) で表されている．MC では *Doloop, Genloop* ノードの構造体に含まれている． ($Narafrase \longrightarrow MC$) *Floop* または *Gloop* の下階層にある *Expression* ノードの条件式の情報から，Doloop もしくは，Genloop を生成する．*Entry* ノードと *Exit* ノードは *Start* ノードと *Stop* ノードに変換する．その他のノードは対応するノードに変換する． ($MC \longrightarrow Narafrase$) MC の *Doloop* または *Genloop* の構造体にあるループ範囲情報から iteration が定数であれば *Floop* ノード，それ以外は *Gloop* ノードとして生成する．*Doloop* または *Genloop* ノード構造体の中のループ範囲情報，またはループ継続条件から *Expression* ノードを生成する．*Doloop* または *Genloop* の下位層にあるノードについては，*Floop* または *Gloop* ノードの下位層に，まず *Block* ノードを生成し，さらにその下位層に対応するノードを生成していく．