TM-0106

ACT^{-1}研究報告集（昭和59年度）
—Theoretical Computer Architecture—

ICOT Working Group WG5

April, 1985

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

ACT^{-1} 研究報告集（昭和５９年度）　ICOT Technical Memo

目次

1.	概要	
2.	関数型言語QUIEの特徴	佐藤　雅彦、桜井　貴文
3.	結合子を用いた簡約化と閉包を用いた簡約化の比較検討	井田　哲雄
4.	Eqlog について	上田　和紀
5.	リダクションシステムからながめた計算モデル	外山　芳人
6.	FPプログラム代数則の証明・手続き	高橋　敦
7.	分散計算モデルと通信計算量	疋田　輝雄
8.	プロセッサ・メモリ結合網のあるクラスの性能評価	萩原　兼一
9.	単一化操作のハードウェアアルゴリズム	安浦　寛人
10.	Read Only AnnotationのUnification とその応用	牧田　正宏
11.	Parallel object oriented Programming with colored Messages	萩谷　昌己

1. 概要

本Research Memo は昭和59年度 ACT^{-1}(Theoretical Computer Architecture) の研究活動の報告集である。 ACT^{-1}はWG5 の中に設けられたグループであり、昭和59年 4月に発足した。本年度の第一の目標は、Theoretical Computer Architecture という、まだ概念のはっきりしない分野における研究の方向付けを与えることであった。そのために、5Gプロジェクトにおいて、身近なしかも具体的な研究として取り組めるテーマと、次世代計算機アーキテクチャを探るより一般的な研究テーマとの関連付けを行うことから活動を開始した。表１の項目が前者の表２が後者の研究課題項目である。 ACT^{-1}の活動指針としては、各構成員の現在および将来の研究テーマと、表１あるいは表２の研究テーマとの一致がみられる部分について、より積極的に、 ACT^{-1}の中で議論を行い、研究の進展を促進させることを試みた。

本報告集にまとめられているのは、以上のような考え方に基づき、行われた12回の委員会の討論の結果である。もとより、表１、２に上げた研究テーマは短期間に片づくものではなく、これからも長期にわたって取り組んでいくものである。しかしながら、１年間の活動内容をまとめておくことは、今後の研究を進展させていく上で、有益であると考え、委員会提出資料に、各委員が再度加筆、整理して、本Research Memo の形にまとめたものである。

表１　研究課題リスト（案、昭和５９年４月）

- KL1 、CPの計算モデル
- QUIEの計算モデル
- データフローとリダクション
- 並列計算の形態とアルゴリズム
- 高階概念の計算実行面からみた評価
- lazy evaluation （遅延評価）の効能と解析
- ユニフィケーションの並列アルゴリズム
- 超並列システム実現上の問題点の検討
- 超並列システムにおけるガーベジコレクション

表2　研究テーマの分類

プログラム言語／プログラミング	計算モデル	計算機アーキテクチャ	素子技術レベル
(1) レベル Programming in the small Programming in the large 仕様記述 (2) データ型 strongly-typed untyped (3) 様式 論理型 関数型 等式（代数） 対象指向 手続型 (4) 構成要素 並列性 高階性 状態遷移 様相 (5) 方法論	(1) 理論体系 ラムダ計算 （結合子論理） 述語論理 （一階，ホーン節） 項書換 ACTOR モデル unification Turing Machine (2) 制御／データ構造 data flow reduction control flow －－－－－－－－－ stack (SECD) heap モデル queue モデル RAM モデル －－－－－－－－－ continuation passing stream (3) 通信形態 メッセージ交換	(1) 動作 逐次型 並列型 (2) レベル 高級言語指向 RISC (3) 並列形態 パイプライン マルチー プロセッサ (4) 物理形態 トポロジー ジオメトリー (5) network 機能分散 （超）並列 systolic array (6) 記憶階層 仮想記憶 データベース	(1) 記憶 連想記憶 マルチー ポート記憶 構造記憶 (2) プロセッサ 専用 はん用 (3) オペランド構造 可変 固定 (4) 同期メカニズム synchronous asynchronous

2. Qute: A Functional Language Based on Unification

Masahiko Sato, Takafumi Sakurai

2.1. Overview

In the paper [4], we have introduced a new programming language called Qute, and defined its semantics formally. The name Qute may be confusing to some readers, since we had already reported about a previous version of Qute in [3]. The new Qute, which we have described in [4], is rather different from the previous one although it inherits many things from them. In spite of this, we have decided to call the new language also Qute.

Qute is a functional programming language which permits parallel evaluation. While most functional languages uses pattern matching as basic variable-value binding mechanism, Qute uses unification as its binding mechanism. Since unification is bidirectional, as opposed to pattern match which is unidirectional, Qute becomes a more powerful functional language than most of existing functional languages.

This approach enables the natural *unification* of logic programming language and functional language. In Qute it is possible to write a program which is very much like one written in conventional logic programming language, say, Prolog. At the same time, it is possible to write a Qute program which looks like an ML (which is a functional language) program.

Qute is a programming language that evaluates an expression under a certain environment which keeps the value of a variable. The evaluation process can be considered as a reduction process of the given expression, and the evaluation stops when the expression has been reduced to a *normal expression* for which no more reduction is possible. Through the process of evaluation the given environment is also changed to another environment by unification. Therefore the result of an evaluation can be considered as a pair of a normal expression and an environment.

In the design of Qute we tried to minimize the number of basic concepts, so that the language becomes easy to learn and specify. These concepts were selected from logical considerations, and as a result some of them were inherited from Concurrent Prolog ([5]) and ML ([2]). In particular, we imported the concepts of 'parallel and' and 'sequential and' from Concurrent Prolog. (However, the current version of Concurrent Prolog does not have 'sequential-and'.)

An expression can be evaluated in parallel (and-parallelism only) and the same result is obtained irrespective of the particular order of evaluation. This is guaranteed by the Church-Rosser property enjoyed by the evaluation algorithm. Although it is possible to add a nondeterministic feature to Qute orthogonally, this point is not discussed in [4], that is, Qute described in [4] does not have 'parallel-or'. Instead, it has 'if-then-else'. The semantics of 'if-then-else' is defined so as to be logical, that is, satisfy the Church-Rosser property. To meet the requirement, the if-part of 'if-then-else' (which can be regarded as a guard) should be evaluated without affecting the environment. As a result, it acts as a synchronization mechanism in evaluating 'parallel-and', which is similar to the synchronization mechanism of Parlog [1]. Therefore, the evaluation of an expression may suspend just as in the case of Parlog or Concurrent Prolog.

2.2. Examples

In this memo, we do not define the semantics of Qute but give two examples which show the expressive power of Qute. Both programs implement the Eratosthenes' sieve by a stream programming technique, but Example 1 is an ML style program and Example 2 a Concurrent Prolog style program.

Example 1

```
primes() <= outstream(sift(integers 2)).
integers n <= [n . (λ (). integers(#n + 1))].
sift [p . s] <= [p . (λ (). sift(filter(#p, #s())))].
filter(p, [n . s]) <=
        if mod(#n, #p) = 0 then
                filter(#p, #s())
        else
                [n . (λ(). filter(#p, #s()))]
outstream [n . s] <= write n; outstream(s())
```

This example shows the basic constructors of Qute: parallel-and (.), sequential-and (;), list (dotted pair), function (λ notation), unification (=), application, and if-then-else. Those who know ML will find that Qute inherits some notations from ML. In this example, stream is represented by a list of an integer (an element of a stream) and a function which produces a tail of a stream when an argument () is supplied. Note that a function is a normal expression and can be passed as an argument of a function without causing a funarg problem.

Example 2

```
primes() <= integers(2, s), sift(s, t), outstream t.
integers(n, [n . s]) <= integers(n + 1, s).
sift(s, t) <=
        if #s = [p . s1] then
                #t = [p . t1]; filter(p, s1, r), sift(r, t1)
        else
                fail
filter(p, s, r) <=
        if #s = [n . s1] then
                if mod(#n, ##p) = 0 then
                        filter(##p, #s1, ##r)
                else
                        #r = [n . r1]; filter(#p, s1, r1)
        else
                fail.
outstream s <=
        if #s = [n . s1] then
                write n; outstream s1
        else
                fail
```

This is an example of the evaluation of parallel-and and the synchronization mechanism. In this example, stream is represented by a list whose tail is undefined, that is, an incomplete data structure. integers(2,s), sift(s,t), outstream t in the definition of primes and filter(p,s1,r), sift(r,t1) in that of sift are evaluated in parallel and the evaluation of if-part (e.g. #s = [p . s1], #s = [n . s1]) succeeds and then-part is evaluated only if non-local variables of if-part (e.g. s) are not instantiated by the evaluation of if-part; otherwise, just before the variables are instantiated the evaluation of if-then-else suspends and waits until they are instantiated through the evaluation of some other expression. (Note: Formal semantics of Qute does not necessarily imply fair evaluation. For example, consider the evaluation of primes(). It is first reduced to three processes integers(2,s), sift(s,t) and outstream t. An implementation which continues to reduce integers(2,s) and never reduce the other two processes is a correct implementation. Actual implementation of Qute provides fair evaluation and it tries to reduce all the three processes little by little.)

2.3. Implementation

A translator of a Qute program into a Prolog program has been implemented under TOPS-20 on DEC2060. The translator is written in DEC-10 Prolog.

References

[1] Clark, K. and Gregory, S., 1984: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Imperial College.

[2] Gordon, M., Milner, R., and Wadsworth, C., 1979: *Edinburgh LCF, Lecture Notes in Computer Science 78*, Springer-Verlag.

[3] Sato, M. and Sakurai, T., 1983: Qute: A Prolog/Lisp type language for logic programming, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 507-513.

[4] Sato, M. and Sakurai, T., 1984: Qute: A Functional Language Based on Unification, *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, 157-165.

[5] Shapiro, E.Y., 1983: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003.

結合子を用いた簡約化と閉包を用いた簡約化の比較検討

理化学研究所　井田哲雄

1．研究の背景

簡約化(reduction) は計算の対象（これを項と呼ぶ）の間に存在する二項関係である。項を計算の世界の抽象的な構成要素と考えることにより、項を簡約化する様々な戦略を具現する計算機（簡約化機械）のアーキテクチャを探ることが筆者の大きな目標である。その一つのステップとして、項をラムダ計算における項に限定して、簡約化の実現手法を検討した。

ラムダ計算における簡約化（以下、特に限定しなければ簡約化とはラムダ計算における簡約化を指す）の計算機による実行の研究の歴史は比較的古い。よく知られた研究としては　C. Wadsworth　のラムダ項のグラフ表現の簡約化の研究がある[1] 。また初期のLISPの実装化の努力は、簡約化の研究と深いつながりがある（注)。Wadsworth の研究は、以降の簡約化の実行最適化の理論的研究に多くの示唆を与えた。言語やアーキテクチャに近い研究としては、P.Landinの言語ISWIM とそれを実行する仮想計算機SECDマシンがある。ISWIM はラムダ計算法の糖衣(sugared)版と考えられる。Landinの手法はラムダ表現を直接実行するのではなく、ISWIM をスタック（Stack)、環境（Environment)、制御リスト(Control）、ダンプ（Dump，状態の保存）をもつ計算機であるSECDマシンのマシン語に翻訳して、実行しようとするものである。さらに、プログラミング言語やアーキテクチャの研究に大きなインパクトを与えたのは１９７０年の後期に発表されたD.Turnerの仕事である。彼は関数型プログラミングの計算のモデルとしてCombinatory Logic（結合子論理系）に着目した。ラムダ計算において任意の項は結合子ＳとＫによって表現されることはよく知られた事実である。Turnerの方法は関数型プログラムをλ項のまま表現せずに，結合子からなる表現に翻訳する。

そして　簡約化規則

Ｓａｂｃ－＞　ａｃ（ｂｃ）

Ｋａｂ　－＞ａ、ここで、ａ、ｂ、ｃは任意の項、

を基本とする簡約化のくり返しによって、計算を実行することにより単純な簡約化システムが実現できることを示した。（彼はＳとＫのほかに簡約化の能率向上のため、いくつかの結合子を導入している。）

注．ふり返ってみれば、そうであるということである。LISP開発の歴史を綴った論文[2] によると、用語上の類似性（たとえば、関数を表すのにLAMBDA標記法を採用した）にもかかわらず，LISPの初期の作成者は，ラムダ計算法をほとんど意識していなっかたようである。

さらに、彼のシステムでは高階関数の実現が容易にできる、正規化戦略である最左簡約化が実現される等、興味深い機能を備えていた。この最左簡約化は、Lazy cons という遅延データ構成関数と併用することにより、最近の関数型プログラミングパラダイムの一つとして知られる遅延評価計算（lazy evaluation)を実現する。確かにTurnerの方法はエレガントである。しかし、実行効率の上では未知の点があり、より精密な検討を必要とする。Turnerは論文[3] において次のような効率の比較をしている。

1．SECDマシンとTurnerの結合子システム、

2．遅延評価を組込んだSECDマシンとTurnerの結合子システム

さらに、Peyton Jonesは

3．ラムダ項を直接解釈するシステム（この場合、環境と項の対を作る必要があり、この対を閉包（closure)とよんでいる）と、Turnerの結合子システムの性能の比較を行っている。

最近では、武市が

4．いくつかの結合子システムと閉包システムを実装し、性能比較をおこなっている。[4]

これらの性能に関する研究は、実際に計算機上にインプリメントして、性能を測定するという手段をとっているが、筆者らは、より理論的な手段での比較を試みた。実装システムの性能評価に、個々の計算機の性能特性を反映させてしまう可能性できる限り排除して、プログラムの複雑度を把握しておきたいことと、新な計算機アーキテクチを考ていく上で、システム実現の複雑度をより基本的なところでおさえておく必要があるからであるからである。

なお、本報告の詳細は次の論文にまとめられている[5]。 T. Ida and A. Konagaya, Comparison of closure and combinatory reduction schemes, 1984 SSEシンポジウム、京都大学数理解析研究所、ICOT-TR072

2．閉包の簡約化

ラムダ計算に関する基本的用語は既知とする。標記法は文献[6] のものを用いる。

項$N = \lambda x.\ M_1\ M_2$ を本稿の以下の議論では図１のようなラベル付木で表現する

項Ｎａ（Ｎをａに作用させたもの、通常のプログラムにおける、実引数ａの関数Ｎの呼出しに対応する）はβ簡約化により $M_1\ [x := a]\ M_2\ [x := a]$ に変形される。

ここで、われわれにとって興味があるのは、Ｎａの正規形を求めることである。

β簡約化は、項の中に出現する変数に対応する値で置換する操作であるが、この操作を機械的に行う方法としてもっとも単純なのは次の方法である。

例１　代入法

$(\lambda x.\ (\lambda y.\ +\ x\ y))\ 4\ 3 \rightarrow (\lambda y.\ +\ 4\ y)\ 3 \rightarrow +\ 4\ 3 \rightarrow 7$

７が正規形

計算機に上の方法を実現するのは能率が悪いので、次のような方法を用いる。

例２　閉包法

$(\lambda x.\ (\lambda y.\ +\ x\ y))\ 4\ 3$

$\rightarrow$ 【$\lambda y.\ +\ x\ y$、｛（ｘ４）｝】３

→ 【＋ｘｙ、｛（ｙ３）（ｘ４）｝】

→ 7

ここで閉包とは、項と環境の対である。環境とは変数と変数が置換される値（項）の対からなる列である。

上の例では、

環境＝｛（ｙ３）（ｘ４）｝

閉包＝【＋ｘｙ、｛（ｙ４）（ｘ３）｝】

閉包による簡約化では、変数に項の代入を直接行わずに、変数が実際に参照された時にのみ、環境の探索により対応値を得ることになる。

閉包による簡約化は一般には、代入法よりも効率がよい。前者では、簡約化の過程において必要とされる変数に関する処理が相異なる変数の数にのみ比例して増加するのに対し、後者では、項の中に出現する記号の数に比例して増加するからである。

注

閉包は例２でもわかるように、正規形へ至る途中で現われてくるものであり、言語の機能を制限するならば、作る必要がない。例えば関数引数を用いないLISP、FPでは不要である。LISPでは λｘｙ．＋ｘｙはこの表現に２つの引数を与えた時にのみ意味をもつので、環境を作成後、直ちに正規形を求めることができる。FPでは変数は概念的には常に１つであると考えることができる。上の例をFP風に解釈しなおすと

λｕ．＋（1 ｕ）（2 ｕ）　1、2は各々列の１、２番目を選択する関数

である。ｕとしては、引数の列をとる。上の例では〈４、３〉である。この時、関数はすべて一引数関数となり、環境が不要になるからである。その代りに、列の選択関数が現われてくる。しかしながら、関数そのものを計算の対象として、正当に扱おうとすると閉包が表に現れてくる。例えば、項（λｘ．（λｙ．＋ｘｙ））４は、４を加えるという一引数の関数と考えることができる。閉包を用いた簡約化法では、この関数は

【λｙ＋ｘｙ、｛（ｘ４）｝】

という閉包によって表現されることになる。

なお、関数型言語の比較を次の表に参考としてまとめた。

言語	データ型としての関数	遅延評価	実装法
KRC／SASL	完全な実現	実現	結合子のグラフリダクション
ML	完全な実現	標準のものは非実現	固定コードとスタック
COMMON LISP	不完全	なし	関数については閉包、ほかに固定コードとスタック
FP	なし	なし	固定コードとスタック

（注）FPの計算モデルであるFFP では関数apply:<f, x> → f:x があり、高階関数や遅延評価一部のクラスの計算は容易に実現できる。

3．結合子による簡約化

結合子による簡約化では項に次の変換規則を再帰的に適用し結合子列に変換する。

$\lambda x.\ M_1 M_2 \rightarrow S M_1 M_2 \quad x \in M_1$ かつ $x \in M_2$

$\lambda x.\ M_1 M_2 \rightarrow B M_1 M_2 \quad x \notin M_1$ かつ $x \in M_2$

$\lambda x.\ M_1 M_2 \rightarrow C M_1 M_2 \quad x \in M_1$ かつ $x \notin M_2$

$\lambda x.\ M \rightarrow K M$ Mは単純項 かつ $M \neq x$

$\lambda x.\ M \rightarrow I\ M = x$

ここで、$\lambda x.\ Mx$の時は$\lambda x.\ Mx \rightarrow M$ （η規則）のように変形して上の変換アルゴリズムを適用することができるが、とりあえずこの最適化は考えない。

そして結合子よりなる項は次の簡約化規則によって、正規形へと簡約化される。

Sabc → (ac) bc

Babc → a (bc)

Cabc → (ac) b

Kab → a

Ia → a

この一連の変換、簡約化は図1と同様のラベル付木を考えることによって直観的意味を与えることができる。（図2－4）

つまり、図2において、

M_1、M_2に自由なxが出現するならば（x M_1、x M_2）λxをSに、

$x \notin M_1$ $x \in M_2$ ならばBを、

$x \notin M_1$ $x \in M_2$ ならばCを、

λxのかわりにらラベルとしてつける。この操作は各部分木についても再帰的に行う。

簡約時には、値を各節に付いたS、B、Cのラベルにしたがい分配し、あらたにできあがった木を簡約化することによっておこなう。（図4）

4．閉包簡約化法のラベル付木による解釈

結合子による簡約化では、簡約化はS、B、Cを除いた定数よりなる木を作り、この木を定数固有の簡約化規則（例＋では2つの数を簡約化して和を得る）による簡約化という操作と理解することができる。これに対応する閉包簡約化ではラベル付木に環境を付加しながらtraverseして、木に「実をならせ」、最終的には、結合子の簡約化と同じく、葉に付いた定数の簡約化規則で、

正規形へと簡約化する。（図５）

５．結合子簡約化法と閉包簡約化法の比較

簡約化に伴う主要な費用は

閉包簡約化においては　環境および閉包の作成

結合子簡約化においては変数を代入値に置き換えた新な木の作成である。

ここで、比較のため次のような仮定をもちこむ。

（１）上記費用は、この簡約化にともなうデータ構造の構築量に比例する、

（２）閉包簡約化については、

2-1 閉包の作成は２セル必要とする。

2-2 環境の作成は、変数一つにつき４セル必要とする。

（３）結合子簡約化については

Ｓについては２セル、

Ｂについては１セル、

Ｃについては１セル必要とする。

（上記数は具体的インプルメンテーションに由来する。ここでは、その妥当性については論じない。）

考察１

簡約化 $(\lambda x.\ M_1\ M_2)\ a \rightarrow M_1[x:=a]\ M_2[x:=a]$ に伴う費用は結合子簡約化法では木の根元の節についたＳ、Ｂ、Ｃのラベルに応じて、各々２、１、１セルである。これにたいして　閉包簡約化法では４である。

考察２

$\lambda x.\ M_1\ M_2 \ldots M_n$、各$M_i$は$\lambda$をふくまない単純項を考える。

簡約化のための木をtraverseに要する費用は２ｎ＋４セルである。ここでｎは簡約化の過程でtraverse　する節の数である。

閉包簡約化においては、木をinorder でtraverseするならば左下進行方向に向っては閉包をつくる必要がないからである。

考察３

閉包簡約化において不要な閉包作成をさけるために、節にＳ、Ｂ、Ｃに対応した最適化情報をラベルとして付けることができる。これを、Ｓ、Ｂ、Ｃ、に対応させてｓ、ｂ、ｃとしよう。

各々の意図するところは次のようなことである。

ｓ　．．．左右部分木の簡約化に環境が必要、

ｂ　．．．右部分木の簡約化に環境が必要、

ｃ　．．．左部分木の簡約化に環境が必要、

こうすると、閉包簡約化においても、セルの消費をおさえることができる。

閉包作成の費用はｓで２セル、ｂで２セル、ｃで０セルになる。

このように考えていくと、結合子簡約化も閉包簡約化も簡約化に要する費用はほぼ同じようなものになることが予想される。

本稿では詳しい説明は省略するが$\lambda x_1 \ldots x_m . M$のような
m変数関数を取扱う時には

結合子簡約化では平均O（$k^{1.5}$）kは葉の総数、

閉包簡約化ではO（m＋log k）の費用がかかることが判明している。しかし、閉包簡約化法では、単純な、LISPのAリストに準じた方法により環境を実現するならばO（m）のアクセスコストがかかるため、必ずしも閉包簡約化のほうが有利とはいえない。コンパイルによって変数の位置をあらかじめ決定することが必要である。この時、各変数名は不要になるので環境生成に要するコストは各変数当り2セルである。以下の表の例では、費用の計算にこの数字を用いている。つぎに具体例について数字を示す。

```
factorial = λn.if (eq n 0) 1 (times n (factorial (- n 1))
foldr = λfkx.if (null x) k (f (hd x) (foldr f k (tl x)))
twice = λfx.f(fx)
```

	閉包	結合子
factorial n!	6n+4	12n+6
foldr 長さn のリスト	14n+7 10n+9（変形）	24n+9 11n+15（変形）
twice 1回につき	4	5 3 （η規則適用）

6．結論

全般的な効率に関して

- ○ 結合子簡約化法、閉包簡約化法ともに全般的な効率は大きな相違がないものと考えられる。ただし、対象プログラムにより効率の違いがある。η規則が適用可能な時は、結合子簡約化法では対象プログラムを表現する木が最初から小さくなっているからである。
- ○ 両者ともに実現上はまだ工夫の余地がたくさんある。
  たとえば、野下－疋田のＢＣ簡法[7]、武市のグラフ複製法[4]などの改良がある。

今後の高速化

- ○ 閉包簡約化にせよ、結合子簡約化法にせよ、高速化のためにはコンパイル（固定コードと動的に変化するデータとの分離）が必要となる。前者のほうが、おそらくコンパイルは容易であろう。
- ○ 現行のLISPのように解釈系と翻訳系の両方を言語処理系にもたせ、解釈系は原始コードに近い表現を閉包簡約化法で、翻訳系は閉包のコンパイル化による高速化のほうが、解釈系の最適化による高速化を図る結合子法よりも、実際的ではないかと思われる。
  特に高速マシンの構築を目ざすならば結合子法はあまりよくないであろう。
- ○ 武市の測定例ではコンパイルした閉包簡約化法はTurnerの結合子簡約化法よりも倍早いことが判明している。

並列処理

- ○ 並列実行環境下では、結合子の部分木、閉包ともにいわゆる参照透明性がなりたつので両者の優劣の差はないものと思われる。

引用文献

[1] C. Wadsworth, Semantics and Pragmatics of the lambda-calculus
Ph. D thesis, University of Oxford, 1971

[2] H. Stoyan, Early LISP history,
Conference Record of LISP and Functional Programming
299-310, Aug. 1984

[3] D.Turner, A new implementation technique for applicative languages, Software Practice and Experience, Vol. 9, 31-39, 1979

[4] 武市正人　結合子式の評価系の実現について，第一回ソフトウエア科学会大会，1984

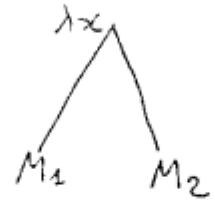
[5] T. Ida and A. Konagaya, Comparison of closure and combinatory reduction schemes, 1984 SSEシンポジウム、京都大学数理解析研究所，ICOT TR072

[6] H.P.Barendregt, The Lambda Calculus:Its syntax and semantics, North-Holland

[7] K. Noshita and T. Hikita, The BC-chain method for representing combinators in linear space, 1984 SSE シンポジウム、京都大学数理解析研究所

図1　　$N \equiv \lambda x.M_1 M_2$ のラベル付木表現

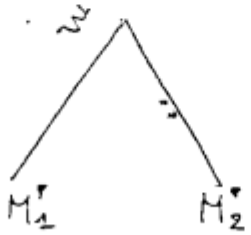
$N \equiv$



λx がラベルである。

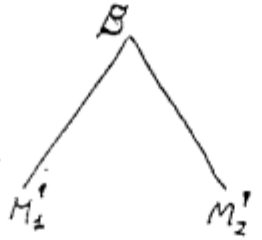
図2　ラベル付木表現の翻訳

2-1



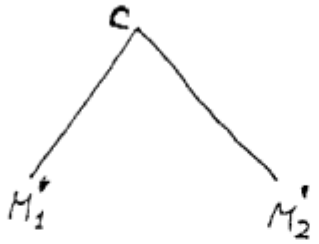
$x \in M_1$, $x \in M_2$

2-2



$x \notin M_1$, $x \in M_2$

2-3



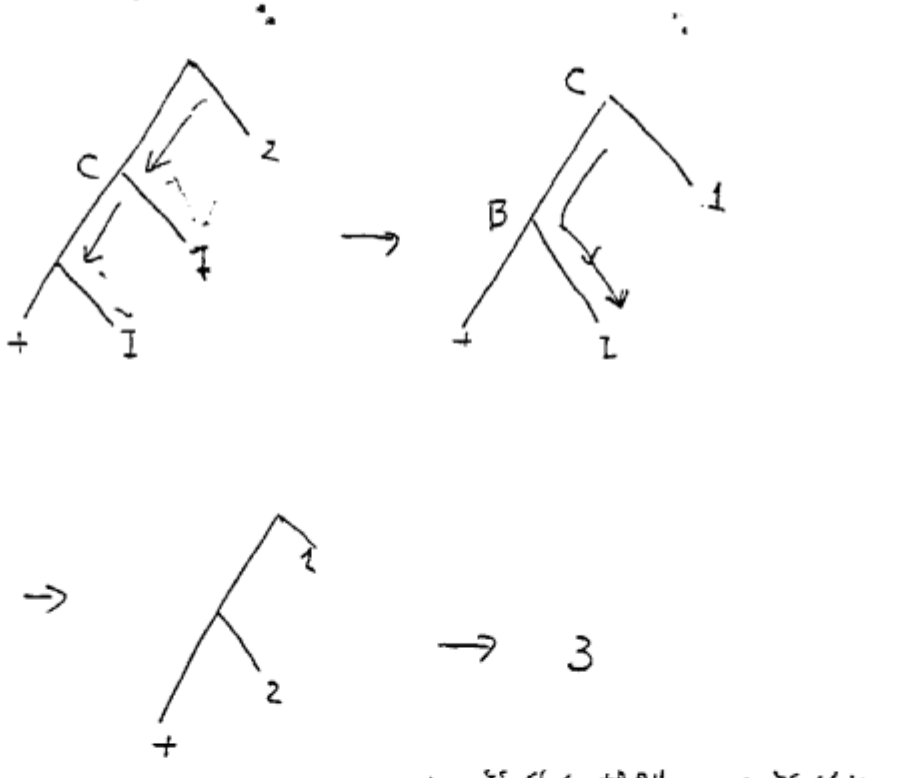
$x \in M_1$, $x \notin M_2$

図3. ラベル付木の解説例

λx.+x 1

図4 ラベル付木の簡約化ー結合子の場合

(λx.+x 1) 2



+の簡約化規則による簡約化

A REVIEW OF EQLOG

March, 1985

Kazunori Ueda
C&C Systems Research Laboratories, NEC Corporation

1. EQLOG = PROLOG U OBJ2

Eqlog [Goguen and Meseguer 84] is a language based on many-sorted first-order logic with equality. Included in Eqlog and not in Prolog are

(1) equality between (syntactically different) terms,
(2) strong typing, and
(3) (generic) modules.

Of these, (2) and (3) are related to many-sortedness, while (1) could stand without many-sortedness.

All the above features are in fact common to OBJ2 [Futatsugi et al. 85] based on equational logic. In particular, (2) and (3) have been introduced in just the same manner as in OBJ2. Therefore, we will focus our attention to how equality is introduced.

2. MANY-SORTED FIRST-ORDER LOGIC WITH EQUALITY

What is different from the ordinary first-order logic is as follows:

(1) (many-sortedness) There exists a concept of sorts and each predicate/function symbol has its arity/rank.
(2) (equality) For each sort 's', a binary predicate '=s' is defined.

In Eqlog, the predicate '=s' is defined by a set of Horn clauses. The equality predicate is called explicitly or implicitly in generalized unification (some researchers call it universal unification). During generalized unification, each clause defining the equality predicate is used as a conditional rewrite rule, where the clause body acts as the condition.

3. GENERALIZED UNIFICATION

The difference of Eqlog and Prolog is that Eqlog uses generalized unification where Prolog uses syntactic unification. In Eqlog, two terms T1 and T2 of the sort s is defined to be unifiable iff

$$C \vdash T1 =_s T2$$

where C is a set of Horn clauses possibly defining '=s'.

In general, there may be infinitary many solutions (i.e., substitution) for a generalized unification problem. Furthermore, complete algorithms which generate all solutions and which explicitly fail in case of no solutions have been found only for

(1) some specific unification problems (e.g., associative-commutative unification), and
(2) a class of unification problems which are described only by

(a) a set R of unit clauses which make up a canonical (i.e., confluent and terminating) term rewriting system, and/or
(b) a set E of unit clauses which make up a specific theory for which finite unification algorithm exists.

For (1) above, Eqlog provides some shorthands (assoc, comm, id, etc.) for specifying theories satisfied by each sort.

Eqlog itself does not seem to limit the class of the equality predicates defined by programmers to one for which a complete set of solutions is guaranteed to exist. However, the authors conjecture that Eqlog's unification algorithm generates a complete set of solutions for an extended class of equality predicate which allows a 'conditional' clause

T1 =s T2 :- B1, B2, ..., Bn.

as long as no Bi uses '=s' directly or indirectly, explicitly or implicitly (during unification).

4. NARROWING

Eqlog uses narrowing in generalized unification. Narrowing is also called paramodulation by other researchers. Narrowing is a repetitive application of one-step narrowing. One-step narrowing rewrites T to T' by

(1) finding a non-variable subterm T0 of T and a rewrite rule T1=T2 such that T0 and T1 are syntactically unifiable by the mgu S, and by
(2) replacing T0 in T by T2 and applying S to T.

Narrowing is like SLD-resolution in that it is a unify-and-rewrite operation.

The generalized unification of T1 and T2 is done as follows:

(1) Make a term e(T1, T2) where e is a new function symbol which can be construed as a formal equality symbol.
(2) Make a narrowing tree whose root is e(T1, T2) and whose paths represent all possible narrowing chains.
(3) For each node e(U, V), check if U and V are syntactically unifiable and if so, compose their mgu with all the substitutions obtained at the narrowing steps from the root down to this node, and add the result to the set of solutions.

If the equality clauses contain a subset E described in Section 3, 'syntactic unification' in the above procedure must be replaced by E-unification throughout.

5. DISCUSSIONS

A useful programming language must have a clear semantics and high efficiency. In this light, the major problem of Eqlog is its unification algorithm. Narrowing has not been implemented in a programming language context as the authors says. However, it is not at all clear whether an efficient and general narrowing algorithm exists; there may be more than one redexes and each narrowing chain may be nonterminating. The generalized unification algorithm and the resolution mechanism must interact with each other. Moreover, in order

to ensure completeness which the authors pursue, the evaluation mechanism of sequential Prolog is inadequate.

Another problem from the programming-language point of view is that of encapsulation.

Equational (algebraic) specification of an abstract data type often serves as the simplest implementation by regarding the specification as rewriting rules. However, one may sometimes want to explicitly give an implementation for the sake of efficiency. Eqlog seems to support this, but not the separation of the specification and the implementation.

For example, suppose we want to use lists to represent data of some type. Then we have to equate the result of some operation with its representation because the abstract operation should immediately be translated to its implementation. However, since the equated values belong to the same type, we have no means to encapsulate the implementation.

A more elegant way to specify implementation would be to describe a separate module giving an implementation and then relate it to the specification module. This could be achieved by supporting the 'derivor' notion [Goguen et al. 78], which is not yet considered in the Eqlog framework.

Eqlog might also be regarded as a specification language. A problem with Eqlog as a specification language might be whether it is truly more expressive than an equational language like OBJ2.

The paper did not convince me that the algorithm for generalized unification (Theorem 2) is sound in the many-sorted case as well as in the one-sorted case. It is well known that the usual deduction rules of one-sorted equational logic are not sound for the many-sorted case [Goguen and Meseguer 81]. However, the description of Theorem 2 is common both to one- and many-sorted cases. The paper should have contained a proof of (un)soundness.

REFERENCES

[Goguen et al. 78] Goguen, J.A., Thatcher, J.W. and Wagner, E., An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, In: (R. Yeh, ed.), Current Trends in Programming Methodology, Prentice-Hall, Englewood Cliffs, NJ, pp.80-140 (1978).

[Goguen and Meseguer 81] Goguen, J.A. and Meseguer J., Completeness of Many-sorted Equational Logic, Sigplan Notices, Vol.16, No. 7, pp.24-32 (1981).

[Goguen and Meseguer 84] Goguen, J.A. and Meseguer J., Equality, Types, Modules and (Why Not?) Generics for Logic Programming, J. of Logic Programming, Vol.1, No.2, pp.179-210 (1984). Also in Proc. Second International Logic Programming Conference.

[Futatsugi et al. 84] Futatsugi K., Goguen, J.A., Jouannaud, J.-P. and Meseguer, J., Principles of OBJ2, to appear in Conf. Record of 12th Annual ACM Symp. on Principles of Programming Languages (1985).

リダクションシステムからながめた計算モデル

外山　芳人

1．はじめに

プログラミング言語の意味をリダクションシステムの概念をもちいて操作的に解釈する。このとき、プログラムの表現している計算過程はすべてリダクションとみなすことが可能となる。プログラミング言語の特徴のいくつかはリダクションシステムの上に抽象的な形で投影される。この抽象的な形で投影された特徴を比較することで、異なるタイプのプログラミング言語の差異を明確にすることが可能となるかもしれない。以下では、リダクションシステムという物差しをもちいて、関数型、論理型、ノイマン型のプログラミング言語を比較する。物差しの目盛りはあらいが、各言語の特徴のいくつかはうまくとらえる。

2．リダクションシステム

まず物差しとしてもちいる抽象的なリダクションシステムについて説明する。

リダクションシステムは$R=\langle D, \rightarrow \rangle$で表わされる。ここでDは対象の集合、→はD上の2項関係でリダクション関係と呼ばれる。→の反射・推移閉包を$\overset{*}{\rightarrow}$で表わす。x、y∈D、$x \overset{*}{\rightarrow} y$ならばxはyにリダクションされるという。xが正規形であるとは$\neg \exists y [x \rightarrow y]$が成立することである。正規形の集合は$NF \subseteq D$によって表わす。$x \overset{*}{\rightarrow} y$、y∈NFならyはxの正規形であるという。

さまざまなプログラミング言語の特徴をリダクションシステムの上で比較する場合、リダクションシステムが以下の特質をもつか否かに注目する。

- リダクションシステムRが停止性をみたすとは、$x1 \rightarrow x2 \rightarrow \cdots$なるリダクションの無限列が存在しないことである。
- Rが弱停止性をみたすとは$\forall x \in D\ \exists y \in NF [x \overset{*}{\rightarrow} y]$が成立することである。
- Rが合流性(Church-Rosser性)をみたすとは
  $\forall x, y, z \in D [x \overset{*}{\rightarrow} y \wedge x \overset{*}{\rightarrow} z \Rightarrow \exists w \in D, y \overset{*}{\rightarrow} w \wedge z \overset{*}{\rightarrow} w]$
  が成立することである。（図1）

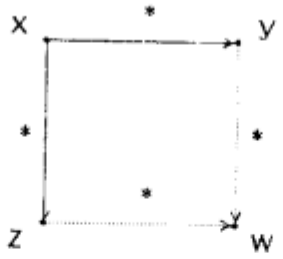


図１．合流性

・Ｒが正規形の一意性をみたすとは、各対象の正規形は高々一個しか存在しないことである。

・Ｒが決定的であるとは、リダクション関係→が（部分）関数となっていることである。そうでないならＲは非決定的という。

３．言語のモデル

関数型プログラミング言語、論理型プログラミング言語、ノイマン型プログラミング言語のモデルとして、FP、LP、NPというリダクションシステムをそれぞれ考える。

・FP

FPは項書き換えシステムである。対象の集合Ｄは項の集合となる。リダクション関係→は書き換え規則によって定められる。書き換え規則はＡ≻Ｂという形をしており、適当な代入θによってＡθが対象の一部分と一致したならば、その一部分をＢθに書き換える。

［例］

書き換え規則：　f(x) ≻ 2×x ,
2× 0 ≻0.
2× 1 ≻2.
2× 2 ≻4.
・・・

リダクションの一例：

f(f(3)) →f(2 ×3)→f(6)→ 2× 6→12.

・LP

LPはリゾリューションシステムである。対象の集合Ｄはゴール節?-C1, C2, …, Cm の集合となる。リダクション関係→は公理を表わすホーン節A:-B1, …, Bn によって定め

られる。対象に書き換え規則としてA:-B1,…,Bn を適用した結果については通常のリゾリューションシステムの推論に従う。

［例］

書き換え規則：　P(x):-Q(x,y),R(y).
　　　　　　　　,Q(x,c).
　　　　　　　　・・・

リダクションの一例：

?-P(a)→?-Q(a,y),R(y) →?-R(c).

・NP

NPは表書き換えシステムである。対象の集合Dは表の集合となる。リダクション関係→は書き換え規則A:=E;(PC:=E') によって定められる。これは表の中のAとPCの値をそれぞれE，E´に書き換えることを意味する。PCは次に適用すべき書き換え規則を一意に定める。

［例］

書き換え規則：　1)　B:=A+B;(PC:=2)
　　　　　　　　2)　A:=A-1;(PC:=3)
　　　　　　　　3)　if A>0 then (PC:=1) else (PC: =4);
　　　　　　　　4)　stop.

リダクションの一例：

A	2
B	3
PC	1

→

A	2
B	5
PC	2

→

A	1
B	5
PC	3

→

A	1
B	5
PC	1

→

A	1
B	6
PC	2

→

A	0
B	6
PC	3

→

A	0
B	6
PC	4

.

各計算モデルは対象の集合D、リダクション関係→の定め方がそれぞれ異なっている。FPではDは項（木）の集合であり、→は書き換え規則とのパターンマッチングによって定められる。LPではDは節の集合であり、→は書き換え規則とのユニフィケーションによって定められる。NPではDは表の集合であり、→は表の中のPCの値によって一意に定めら

れる。しかし、抽象的なリダクションシステムの上では、これらの差異は、リダクション関係→の性質として特徴付けられる部分のみが具体的に現われる。このような抽象化をおこなっても、各言語の本質的な特徴のいくつかは明確な形で残ることを表１で示す。

表１．それぞれの計算モデルの特徴

	ＦＰ	ＬＰ	ＮＰ
何を計算の答とみるか？	正規形	正規形の特別な形（空節口等）	正規形
リダクションの決定性	非決定的	非決定的	決定的
リダクションの停止性	プログラムが意味をもつためには弱停止性が必要	プログラムが意味をもつためには弱停止性が必要	プログラムが意味をもつためには強停止性が必要
答の一意性	プログラムが意味をもつためには正規形の一意性が必要。一般には、より強い性質である合流性をみたす。	正規形の一意性、合流性は一般には成立しない。したがって答は複数個存在する場合がある。	リダクションが決定的であるので一意性は保証されている。

表１では対象の集合Ｄの構造とリダクション関係→の定め方をすべて無視して、各計算モデルの特徴付けをおこなっている。したがって、この特徴付けに従えばユニフィケーションをもちいてリダクション関係→を定める計算システムであっても、それが抽象的なリダクションシステムとして、たとえば正規形の一意性や合流性などの特徴をもつならばFP型と見なされる。また、Ｄとして項の集合をもちい、パターンマッチングによってリダクション関係を定めるシステムであっても、合流性が保証されておらず答が複数個存在するならばLP型になる。

このような視点は、新しい計算システムを考える場合には参考になると思う。関数型と論理型の両者の特徴をかねそなえた計算システムでは、Ｄの構造や→の設定方法のみでシステムを特徴付けることが困難となるからである。

なお、ＦＰ、ＬＰ、ＮＰのより詳細な比較には、さらに具体的な構造をもつリダクションシステムの物差しが必要となる。どのようなリダクションシステムに投影すべきかは今後の問題である。

A Proof Procedure for the Algebra of FP programs

Atsushi TOGASHI, Hiroshi NUNOKAWA and Shoich NOGUCHI

Research Institute of Electrical Communication, Tohoku University
2-1-1 Katahira-cho, Sendai-shi 980 Japan

1. INTRODUCTION

In his Turing Award Lecture (Backus 78) and subsequent papers (Backus 81a,81b) John Backus promoted a "functional style" of programming with an associated algebra of programs as an alternative to the conventional "word-at-a-time oriented Non Neumann style" of programming. Programs in this FP style are built from a set of primitive programs by a small set of program forming operators (combining forms) and by recursive definitions. The FP programs have extremely simple and clear semantics; the principal program forming operators are operations of a powerful algebra of programs that can be used to reason about and understand FP programs.

The present paper is concerned with a mechanical proof procedure for algebraic laws in the FP programs algebra. In his lecture (Backus 78) Backus illustrated some useful algebraic laws, some of them require a certain extent of consideration on programs to prove them. We propose a new mechanical proof procedure for the program algebra, which is an extension of the Knuth-Bendix completion algorithm (Huet 81). This provides us a mechanical reasoning about algebraic laws whose proof usually requires induction on objects or functions.

In chapter 2, we will briefly survey preliminary definitions such as sorted algebras, algebraic specifications, term rewriting systems and so on. The groundwork which will form the basis of this paper is also discussed. In chapter 3, we specify a FP system as a term rewriting system as in (Dosch & Möller 84, Möller 84), some properties of the resulting system and its relationship with the final model semantics of the associated specification are

discussed. Finally in chapter 4, we describe the modified Knuth-Bendix completion algorithm to prove the algebraic laws of the FP programs.

2. PRELIMINARY DISCUSSIONS

In the following sections we prepare the groundwork which will form the basis of this paper.

2.1 Sorted Algebras

This section is a brief survey of many-sorted algebras, see (ADJ 78, Huet & Oppen 80, Wand 79). This provides the key to the following discussions. It is assumed that we are given a finite set S of sorts, which are the names of the various objects under consideration. A (S-sorted) signature is an S^*xS-indexed family of disjoint sets $\Sigma_{w,s}$ of symbols, where S^* denotes the set of all finite sequences on S including the null string λ. A symbol $f \in \Sigma_{w,s}$ is called an operation symbol of sort s with rank w, written by $f : w \rightarrow s$. When $w = \lambda$ f is called a constant. For ease of notation, let $\Sigma = U_{(w,s) \in S^*xS} \Sigma_{w,s}$, and we use Σ to denote a signature.

A Σ-algebra (or an algebra in short) is a pair (A, Σ_A), where A is a S-index family of sets A_s, called carriers of sort s, and Σ_A is a Σ-index family of operations such that $f_A : A_{s1} x \ldots x A_{sn} \rightarrow A_s$ whenever $f : s_1, \ldots, s_n \rightarrow s \in \Sigma$. One of the most useful algebras is the term algebra which consists of terms. Now we define a term on the signature.

Let $X = U_{s \in S} X_s$ be a disjoint union of denumerable sets X_s of variables of sort s S. For a signature Σ, Σ-terms (or terms whenever Σ is clear from the context) of sort s are defined in the usual way, see (ADJ 78). The set of all Σ-terms of sort s is denoted by $T(\Sigma,X)_s$. We define $T(\Sigma,X)$ as the disjoint union of the sets $T(\Sigma,X)_s$ for $s \in S$. The corresponding algebra with the usual definition of operations is called a term algebra.

Let A and B be two algebras on the same signature (S,Σ). A morphism from A into B is a S-indexed family of mappings $h_s : A_s \rightarrow B_s$ such that

$$h_s(f_A(a_1,\ldots,a_n)) = f_B(h_{s1}(a_1),\ldots,h_{sn}(a_n))$$

for every $f : s_1,\ldots,s_n \dashrightarrow s$, and for every a_i in A_{si}. A morphism $\theta : A \rightarrow B$ is called a substitution if $A = B = T(\Sigma,X)$. Substitutions are uniquely determined by the values of the variables.

A valuation (of variables in A) is a function $v : X \dashrightarrow A$. Any valuation induces a morphism $v^{\#} : T(\Sigma,X) \dashrightarrow A$ in the obvious way. If t is a ground term, the value of t in A is uniquely determined independent from the valuations. We denote this value by t_A.

An equation is a pair of terms, denoted by $t = t'$. An algebra A satisfies an equation $t = t'$ if $v^{\#}(t) = v^{\#}(t')$ for every valuation $v : X \dashrightarrow A$. A satisfies the set of equations E if A satisfies every equation in E.

2.2 Algebraic Specifications

This section briefly reviews basic notions and properties about algebraic specifications for systems, following (ADJ 78, Musser 78,80).

Definition Given a signature (S,Σ), a specification is a triple $SP = (S,\Sigma,E)$, where E is a set of equations of the same sort.

Any specification induces the congruence relation $\equiv_E$ on $T(\Sigma,X)$. Intuitively, $\equiv_E$ is the congruence relation identifying exactly all terms, the equality of which is deducible from E. If A is an algebra, let $\equiv_A$ denote the congruence relation on terms defined by $t \equiv_A t'$ iff $v^{\#}(t) = v^{\#}(t')$ for every valuation v in A. Given a specification $SP = (S,\Sigma,E)$, an algebra A is SP-algebra if A satisfies E.

Proposition 2.1 Let $SP = (S,\Sigma,E)$ be a specification. An algebra A is a SP-algebra iff $\equiv_E \subset \equiv_A$ holds.

Proposition 2.2 For any specification $SP = (S,\Sigma,E)$, $Im(E) = T(\Sigma)/\equiv_E$ is an initial SP-algebra in the sense that there exists a unique morphism $h : Im(E) \dashrightarrow A$ for every SP-algebra A.

Let $SP_0 = (S_0,\Sigma_0,E_0)$ be the distinguished subspecification of SP. Note that each operation symbol in Σ_0 takes a type in S_0 and every term in $T(\Sigma_0,X)$ takes a sort in S_0.

Definition SP = (S,Σ,E), or E, is (with respect to SP_0)

(1) consistent if and only if $t \equiv_{E0} t'$ whenever $t \equiv_E t'$ for all ground terms t, t' in $T(\Sigma_0)$;

(2) sufficiently complete if and only if for every term t' in $T(\Sigma)_{s0}$, $s_0 \in S_0$, there exists t in $T(\Sigma_0)_{s0}$ such that $t \equiv_E t'$.

Definition A context of type, $s \dashrightarrow s'$, is a term C(x) of sort s' such that C(x) contains at most one variable x of sort s.

Definition Let SP = (S,Σ,E) be a specification with SP_0. We define a relation $\sim_E$ on terms by $t \sim_E t'$ iff $C(t\theta) \equiv_E C(t'\theta)$ for every context $C(x) : s \dashrightarrow s_0$, $s_0 \in S_0$, and for every ground substitution θ.

Intuitively, $\sim_E$ distinguishes exactly those terms which behave differently in some context of the distinguished sort or substitution.

Proposition 2.3 We have the following properties on $\sim_E$:

(1) $\sim_E$ is a congruence relation on $T(\Sigma,X)$.

(2) $\equiv_E \subset \sim_E$.

2.3 Final Model

For the rest of this chapter we assume that the specification SP = (S,Σ,E) is both consistent and sufficiently complete with respect to the specified subspecification SP_0.

Definition An algebra A is an interpretation of SP if A satisfies the following conditions:

(1) For ground terms t, t' in $T(\Sigma_0)$ $t \equiv_{E0} t'$ whenever $t_A = t_A'$.

(2) Every a in A can be represented by some term t in $T(\Sigma)$, i.e. $t_A = a$.

Proposition 2.4 If A is an interpretation, then A and the quotient term algebra $T(\Sigma)/\equiv_A$ are isomorphic.

Definition An interpretation A is a model of SP if and only if A satisfies every equation in SP.

Proposition 2.5 Let A be a model of SP. Then the restriction of A to Σ_0 is an initial SP_0-algebra.

Lemma 2.1 Let A and B be interpretations of SP. A mapping h : A --> B is a morphism iff $h(t_A) = t_B$ for every ground term t.

Proof. The "only if" part is easily verified by structural induction on terms. On the other hand, suppose h : A --> B is a mapping such that $h(t_A) = t_B$ for every ground term. Let $f : s_1,\ldots,s_n \rightarrow s$, and $a_i \in A_{si}$ for $i = 1,\ldots,n$. Since A is the interpretation of SP we can choose a term t_i such that $t_{iA} = a_i$ for each element a_i. By assumption $h(a_i) = h(t_{iA}) = t_{iB}$.

$$\begin{aligned} h(f_A(a_1,\ldots,a_n)) &= h(f_A(t_{1A},\ldots,t_{nA})) \\ &= h(f(t_1,\ldots,t_n)_A) \\ &= f(t_1,\ldots,t_n)_B \\ &= f_B(t_{1B},\ldots,t_{nB}) \\ &= f_B(h(a_1),\ldots,h(a_n)). \end{aligned}$$

So that h is a morphism.

Corollary 2.1 Let A and B be interpretations of SP. Then there exists at most one morphism h : A --> B from A into B.

Proof. Suppose h_1, h_2 are two morphisms from A into B. By the definition of the interpretation we can select a term t for each element a in A such that $t_A = a$. By Lemma 2.1 $h_1(a) = h_1(t_A) = t_B = h_2(t_A) = h_2(a)$. Hence, the result follows.

Theorem 2.1 (Wand 79) If SP =(S, ,E) is a consistent and sufficiently complete specification with respect to SP_0, there exists a final model Fm(E) of SP. Furthermore Fm(E) is characterized in the following way:

A ground equation t = t' holds in Fm(E) if and only if there exist a finite sequence of ground terms $t = t_0,\ldots,t_n = t'$ and a finite sequence of SP models $A_1,\ldots,A_n$ such that $t_{i-1Ai} = t_{iAi}$ for $i = 1,\ldots,n$.

Proof. Let Q denote a relation on $T(\Sigma)$ defined by t Q t' iff there exist a finite sequence of ground terms $t = t_0,\ldots,t_n = t'$ and a finite sequence of SP models $A_1,\ldots,A_n$ such that $t_{i-1Ai} = t_{iAi}$ for $i = 1,\ldots,n$. Q is not, in fact, void since $T(\Sigma)/\equiv_E$ is one such instance of SP models. It is clear from the definition that Q is the least congruence relation including the induced relations $\equiv_A$ from models A. Thus Q satisfies E.

Now we shall show $T = T(\Sigma)/Q$ is the final model of SP. To show that T is an interpretation, suppose s Q t and $s = t_0,\ldots,t_n = t$ is the

corresponding sequence such that $t_{i-1Ai} = t_{iAi}$ for some model A_i, where s and t are in $T(\Sigma_0)$. For each term t_i we can choose a term t'_i in $T(\Sigma_0)$ such that $t_i \equiv_E t'_i$ since SP is sufficiently complete. Hence $t_{iAi} = t'_{iAi}$ holds for i = 1,...n. Let us consider a sequence $s = t_0',\ldots,t_n' = t$. Note that $t_{i-1}'_{Ai} = t_i'_{Ai}$ for each i. Since each t_i' is a term in $T(\Sigma_0)$ and each A_i is a model, $t_{i-1}' \equiv_{E0} t_i'$ for i = 1,...,n. Thus $s \equiv_{E0} t$. So that T is not only an interpretation but a model of SP by Proposition 2.1.

Suppose A is a model of SP. Recall that $A \cong T(\Sigma)/\equiv_A$. Let us define a mapping $h : T(\Sigma)/\equiv_A \rightarrow T$ by $h([t]_{\equiv A}) = [t]_Q$, where t is in $T(\Sigma)$. h is a well defined morphism by Lemma 2.1. Corollary 2.1 implies this is the only morphism from A into T. This completes the proof of the theorem.

Corollary 2.2 Let T = Fm(E) be a final model and A a model of SP. If an equation t = t' holds in A, it holds in T also.

Proof. Suppose t = t' holds in A. To prove the conclusion of the corollary it suffices to show that $\theta(t)_T = \theta(t')_T$ for every ground substitution θ by the construction of the final model. Let θ be any ground substitution and v be the valuation in A defined by $v(x) = \theta(x)_A$. Since A satisfies t = t', $\theta(t)_T = h(v^{\#}(t)) = h(v^{\#}(t')) = \theta(t')_T$, where $h : A \rightarrow T$ is the unique morphism.

2.4. Equational theories in final model

The discussions in the previous section suggest that the equational theory in the final model can be characterized in the similar way.

Let us define a relation ~ on terms: t ~ t' iff there exist a finite sequence of terms $t = t_0,\ldots,t_n = t'$ and a finite sequence of SP models $A_1,\ldots,A_n$ such that $t_{i-1} = t_i$ holds in A_i for i = 1,...,n. ~ is obviously a congruence relation. The direct consequence from Corollary 2.2 is the next Theorem.

Theorem 2.2 t ~ t' iff t = t' holds in the final model of SP.

Proof. The "if" part is obvious from the definition of ~. The "only if" part is verified by Corollary 2.2.

Lemma 2.2 Let $t = t'$ be an equation. In the final model $t = t'$ holds if and only if $t\theta = t'\theta$ holds for every ground substitution θ.

Proof. Since the "only if" part is obvious, we prove the "if" part only. Let us assume that $t\theta = t'\theta$ for every ground substitution θ. Suppose v is a valuation in the final SP model $T = Fm(E)$, then there is a ground substitution θ such that $v(x) = \theta(x)_T$ for each x in X. By assumption $t\theta = t'\theta$, i.e. $\theta(t)_T = \theta(t')_T$. Hence $v^{\#}(t) = \theta(t)_T = \theta(t')_T = v^{\#}(t')$. This means $t = t'$ holds in T.

Lemma 2.3 For ground terms t and t', $t = t'$ holds in the final model T whenever $t \sim_E t'$.

Proof. To prove the lemma it suffices to show that $T(\Sigma)/\sim_E$ is an interpretation. Suppose there are two ground terms t, t' in $T(\Sigma_0)$ such that $t \not\equiv_{E0} t'$, however for every context $C(x) : s \dashrightarrow s_0$, $C(t) \equiv_E C(t')$. By taking $C(x) = x$ as a context, $C(t) = t \not\equiv_E t' = C(t')$ since SP is consistent. This contradicts the assumption. Therefore we have obtained the result of the lemma.

Theorem 2.3 For terms t and t', $t = t'$ holds in the final model if and only if $t \sim_E t'$.

Proof. Let t, t' be terms. Suppose $t = t'$ holds in the final model. By Theorem 2.2 suppose $t = t_0, \ldots, t_n = t'$ and $A_1, \ldots, A_n$ are the corresponding sequences of terms and SP models respectively such that $t_{i-1} \equiv_{Ai} t_i$ for $i = 1, \ldots, n$. Let $C(x) : s \dashrightarrow s_0$ be any context and θ be any ground substitution. Since A_i satisfies $t_{i-1} = t_i$ it also satisfies $C(t_{i-1}\theta) = C(t_i\theta)$. Note that both terms $C(t_{i-1}\theta)$ and $C(t_i\theta)$ are ground terms of the sort s_0 in S_0. Without loss of generality, we can assume that the both terms $C(t_{i-1}\theta)$ and $C(t_i\theta)$ belong to $T(\Sigma_0)$ since SP is sufficiently complete. These considerations imply that $C(t_{i-1}\theta) \equiv_{E0} C(t_i\theta)$. Hence, $C(t_{i-1}\theta) \equiv_E C(t_i\theta)$.

On the other hand suppose $t \sim_E t'$ and let θ be any ground substitution. Then we have $t\theta \sim_E t'\theta$. By Lemma 2.3 $t\theta = t'\theta$ holds in the final model. So that the proof is complete by Lemma 2.2

2.5 Term rewriting systems

Definition A term rewriting system (on the signature Σ) is a set of rewriting rules R such that $Var(l) \supset Var(r)$ for every rule $l \to r$ in R.

The reduction relation $\to_R$ associated with R is the finest relation over $T(\Sigma, X)$ containing R and closed under the operations substitution and replacement. That is

(1) $M \to_R N$ implies $M\theta \to_R N\theta$;
(2) $M \to_R N$ implies $P(u \leftarrow M) \to_R P(u \leftarrow N)$.

Equivalently, $\to_R$ is the finest relation containing all pairs $l\theta$, θr such that $l \to r$ in R and closed under replacement (2).

From now on, we shall use $\to$ for $\to_R$. We use the standard notations $\overset{+}{\to}$ for the transitive closure of $\to$, $\overset{*}{\to}$ for its transitive, reflexive closure, and $\leftrightarrow$ for its symmetric closure. Note that $\overset{*}{\leftrightarrow}$ is the same as the congruence relation $\overset{=}{}_R$, when R is considered as a set of equations.

Definition R is Church-Rosser (or R has a Church-Rosser Property) if and only if for all M and N, $M \overset{=}{}_R N$ if and only if there is a P such that $M \overset{*}{\to} P$ and $N \overset{*}{\to} P$.

An equivalent characterization is "confluence".

Definition R is confluent if and only if for all M, N and P $P \overset{*}{\to} M$ and $P \overset{*}{\to} N$ implies there is some Q such that $M \overset{*}{\to} Q$ and $N \overset{*}{\to} Q$.

We say that M is irreducible or in normal form (relative to R) if there is no N such that $M \to N$, that is, no subterm of M is an instance of some left hand side of a rule in R. We say that N is a normal form of M, denoted by M!, iff $M \overset{*}{\to} N$ and N is a normal form relative to R. When R is confluent the normal form is unique, if it exists. A sufficient condition for existence of such a normal form is the termination of all rewritings:

Definition R is Noetherian (or finitely terminating) iff there is no infinite reduction sequence.

Let $l \to r$ and $l' \to r'$ be two rules in R. Assume u is a non-variable occurrence in l such that l/u and l' are unifiable with a mgu θ. We say that the pair $(l(u \leftarrow r')\theta, r\theta)$ is critical in R.

To sum up the results obtained in this chapter, we have

Theorem 2.4 Let SP = (S,Σ,E) be a consistent and sufficiently complete specification with respect to the specified SP_0 and Fm(E) the final model of SP. Suppose t, t' are terms, then the following claims are equivalent.

(1) $t \sim_E t'$: $C(t\theta) \equiv_E C(t'\theta)$ for every ground substitution θ and for every context C(x): $s \dashrightarrow s_0$, s_0 in S_0.

(2) $t \sim t'$: there exist a sequence of terms $t = t_0,\ldots,t_n = t'$ and a sequence of SP models $A_1,\ldots,A_n$ such that $t_{i-1} \equiv_{Ai} t_i$, for $i = 1,\ldots,n$.

(3) $t = t'$ holds in Fm(E).

(4) $t\theta = t'\theta$ holds in Fm(E) for every ground substitution θ.

3. FP SYSTEM AS A TERM REWRITING SYSTEM

In this chapter, we specify the Backus' FP system as a term rewriting system as in (Dosch & Möller 84, Möller 84). Operation symbols are partitioned into two types, constructors and extenders. Constructors are used to create object / function terms in the system and extenders to define manipulations on the constructed objects, the meaning of them are specified as rewriting rules.

3.1 FP system FUNC

The FP system employs a variety of atomic data types such as Boolean values, natural numbers, characters and so forth. As an example a simple system for the Boolean values is specified as follows:

```
system BOOL
   sort; bool
   constructor; T, F, !bool : λ --> bool
   extender; NOT : bool --> bool

   rules;
         NOT(T) -> F
         NOT(F) -> T
         NOT(!bool) -> !bool
end system
```

In the sequel we assume that a primitive system ATOM is defined as the disjoint sum of the primitive data types. Based on ATOM a system OBJECT is specified as follows:

```
system OBJECT = ATOM U
   sort; object
   constructor;
         ∅, !_obj : λ --> object
         in : atom --> object
         _&_ : object, object --> object

   extender;
         pro : object --> atom
         hd, tl, null, apndl, apndr,
           distl, distr, id : object --> object
         _ ⇒ _; _ : object, object, object --> object

   rules;
         pro(∅) -> !_atom                pro(!_obj) -> !_atom
         pro(in(a)) -> a                 pro(x & y) -> !_atom

         hd(∅) -> !_obj                  hd(!_obj) -> !_obj
         hd(in(a)) -> !_obj              hd(x & y) -> x

         tl(∅) -> !_obj                  tl(!_obj) -> !_obj
         tl(in(a)) -> !_obj              tl(x & y) -> y

         null(∅) -> in(T)                null(!_obj) -> !_obj
         null(in(a)) -> !_obj            hd(x & y) -> in(F)

         apndl(x & ∅ & ∅) -> (x & ∅)
         apndl(x & (y & z) & ∅) -> (x & y & z)
         apndl(x) -> !_obj               otherwise

         apndr(∅ & x & ∅) -> (x & ∅)
         apndr((x & y) & z & ∅) -> (x & apndr(y & z & ∅)
         apndr(x) -> !_obj               otherwise

         distl(x & ∅ & ∅) -> ∅
         distl(x & (y & z) & ∅) -> ((x & y & ∅) & distl(x & z & ∅))
```

```
        distl(x) -> !_obj                    otherwise

        distr(∅ & x & ∅) -> ∅
        distr((x & y) & z & ∅) -> ((x & z & ∅) & distr(y & z))
        distr(x) -> !_obj                    otherwise

        id(x) -> x

        (in(T) => x; y) -> x          (in(F) => x; y) -> y
        (u => x; y) -> x                     otherwise
end system
```

Note that the usual sequence $x = (x_1,x_2,...,x_n)$ can be represented by the constructor term as:

$$(x_1 \,\&\, x_2 \,\&\, \ldots \,\&\, x_n \,\&\, \emptyset) = (x_1 \,\&\, (x_2 \,\&\, (\ \ldots \,\&\, (x_n \,\&\, \emptyset)\ldots)$$

In the FP system a fixed set of "program forming operators", i.e., combinators, is used for combining the basic functions into complex functional forms. In the sequel we define a system FUNC that describes functional forms without recursion.

```
system FUNC = OBJECT U
   sort; function
   constructor;
         ß, f* (primitive functions) :λ --> function
         ¯ : object --> function
         °, $, # : function, function --> function
         /, @ : function --> function
         --> ; : function, function, function --> function
```

(note ß : empty function; f^* : atomic function corresponding to the primitive function f; ° : composition; $: construction; # : parallel construction; ¯ : constant; -->; : conditional; / : insertion; @ : apply to all)

```
         extender;
               :  : function, object --> object

         rules
               ß:x -> ∅   for x ≠ !_obj
```

```
ß:!_obj -> !_obj
f*:x -> f(x)
x̄:y -> x    for y ≠ !_obj
x:!_obj -> !_obj

(f $ g):x -> ((f:x) & (g:x))
(f # g):(x $ y) -> ((f:x) & (g :y))
(f # g):x -> !_obj                    otherwise
(f ° g):x -> (f:(g:x))
(f -->g;h):x -> ((f:x) ⇒ (g:x); (h:x))

/f:(x  ∅) -> (x & ∅)
/f:(x & y & z) -> f:(x & (/f:(y & z)))
/f:x -> !_obj                         otherwise
@f:∅ -> ∅
@f:(x & y) -> ((f:x) & (@f:y))
@f:x -> !_obj                         otherwise
end system
```

3.2 Discussions on the resulting system

To relate the above system with the Backus' original FP system we call a ground term of sort object, a sequence, if it is constructed only by constructors, i.e. a term in $T(\Sigma_0^C)_{obj}$. Similarly terms of sort object and sort function are called object terms and function terms respectively.

Proposition 3.1 Every ground object term is reducible to a sequence.

Proof. We prove this property by structural induction on ground object terms. Let t be a ground object term.

Basis: If t is $!_{obj}$, ∅, or in(a) for some atom a, t is a sequence.

Induction: (1) Suppose t is of the form $(t_1 \ \& \ t_2)$. As an induction hypothesis, we assume that $t_1 \xrightarrow{*} t_1'$, $t_2 \xrightarrow{*} t_2'$ for some sequences t_1' and t_2'. Thus, $(t_1 \ \& \ t_2) \xrightarrow{*} (t_1' \ \& \ t_2')$, and $(t_1' \ \& \ t_2')$ is a sequence.

(2) Suppose t is of the form $t = pf(t_1)$ or $(t_1 \Rightarrow t_2; t_3)$, where pf is a primitive function symbol except the conditional and each t_i is a ground object term. To verify the result we must show that t is reducible to a sequence whenever each t_i is reducible to a sequence. We shall show this only for the case $t = (t_1 \Rightarrow t_2; t_3)$. Other cases are similar to this case.

Suppose t_1 is reducible to a sequence t_1'. By the previous discussion t_1' must be of the form $\underline{!}_{obj}$, $\emptyset$, in(a) for some atom a, or $(t_{11}' \& t_{12}')$ for some sequences t_{11}' and t_{12}'. Even if t_1' is described in the alternative forms t is reducible to a sequence by the rewriting rules for the primitive function symbols.

(3) Suppose t is of the form $t = f{:}t_1$, where f is a ground function term and t_1 is a ground object term. To complete the proof we must show the following assertion by structural induction on ground function terms:

Every ground object term f:t is reducible to a sequence if t is reducible to a sequence.

This can be easily verified in exactly the same way as in (2), so we omit it.

Proposition 3.2 The FP system described above is a Noetherian and confluent term rewriting system.

Proof. The confluence is obvious, since there is no overlapping on the left hand side of the rules. The Noetherian property can be easily proved by structural induction on object terms in a way similar to the proof of Proposition 3.1.

Let FP = (S,Σ,E) be the specification of the FP system, where E is the set of equations defined by ignoring the orientation of the rewriting rules in the system. Let FP_0 be the one for the system OBJECT. Recall that Σ (Σ_0) is partitioned into the set of constructors Σ^c (Σ_0^c) and the set of extenders Σ^e (Σ_0^e).

$$\Sigma = \Sigma^c \cup \Sigma^e \qquad \Sigma_0 = \Sigma_0^c \cup \Sigma_0^e$$

To establish sufficient completeness and consistency for FP we will prepare the next proposition.

Proposition 3.3 FP satisfies the following conditions (called "the principle definition" investigated in (Huet & Hullot 82)). (For later use we shall state these properties 1,2 more abstractly)

Property 1: For every ground term M in $T(\Sigma_0)$, there exists a constructor term M_0, i.e. a term in $T(\Sigma_0^c)$, such that $M \equiv_{E0} M_0$.

Property 2: For constructor terms M and N, $M \equiv_{E0} N$ only if M = N.

Proof. Property 1 follows from Proposition 3.3. Since there is no rule which rewrites a sequence, Property 2 holds.

We can further strengthen the above properties in the following way:

Property 1': For every ground term M in $T(\Sigma)$, there exists a constructor term M_0, i.e. a term in $T(\Sigma_0^c)$, such that $M \equiv_E M_0$.

Property 2': For constructor terms M and N, $M \equiv_E N$ only if M = N.

Proposition 3.4 FP is consistent and sufficiently complete, so that there exists a final model Fm of FP.

Proof. Sufficient completeness is clear by Proposition 3.1, so we prove only that FP is consistent. Let t, t' be ground object terms in $T(\Sigma_0)$ such that $t \equiv_E t'$. By Proposition 3.2, t and t' can be reducible to the unique normal forms t! and t'! respectively by applying only the rules in OBJECT, since no program forming operator appears in both t and t'. Thus, $t \equiv_{E0} t!$ and $t' \equiv_{E0} t'!$. Since both t! and t'! are sequences, $t! \equiv_E t'!$ implies t! = t'! by Property 2'. Hence, we have $t \equiv_{E0} t'$.

3.3 Algebra for FP programs

Definition Let f = g be an equation of sort function. f = g is an algebraic law in the FP algebra iff $f \sim_E g$.

Lemma 3.1 Let F(u) be a function term which contains at most one variable u of sort function. Suppose f, g are ground function terms such that $f:t \equiv_E g:t$ for every sequence (equivalently for every ground object term), then $F(f):t \equiv_E F(g):t$ for every sequence t.

Proof. By structural induction on F(u).

Theorem 3.1 Let f = g be an equation of sort function. The following three conditions are equivalent.

(1) f = g is an algebraic law, i.e. $f \sim_E g$.

(2) $f\theta:t \equiv_E g\theta:t$ for every ground substitution θ and sequence t.

(3) $(f\theta:t)! = (g\theta:t)!$ for every ground substitution θ and sequence t.

Proof. The proofs (1) => (2) and (2) => (3) follow from the definition of $\sim_E$ and Proposition 3.2 respectively. (2) => (1) can be proved by Proposition 3.1, 3.2 and Lemma 3.1.

To conclude this chapter we will state the main theorem obtained in this chapter which relates the algebraic laws for FP programs with a proof procedure in the final model of FP.

Theorem 3.2 Let $t = t'$ be an equation consisting of function terms t, t'. Then the next alternative assertions are equivalent.

(1) $t = t'$ is an algebraic law.

(2) $t = t'$ holds in the final model Fm of FP.

(3) $t\theta = t'\theta$ holds in the final model Fm of FP for every ground substitution θ.

(4) $t \sim t'$.

(5) $t\theta:x \equiv_E t'\theta:x$ for every ground substitution θ and sequence x.

(6) $(t\theta:x)! = (t'\theta:x)!$ for every ground substitution θ and sequence x.

(7) $C(t\theta) \xrightarrow{*} y$ iff $C(t'\theta) \xrightarrow{*} y$ for every ground substitution θ and for every ground object term y.

4. A PROOF PROCEDURE

In this chapter we present a proof procedure in the algebra of FP programs. This is the modified version of the Knuth-Bendix completion algorithm in (Huet 81). The subsequent discussions give the basis for the proof procedure which will be described later.

Proposition 4.1 Let E, E' be sets of equations (on the same signature) such that E' contains E, that is $\equiv_E$ is contained in $\equiv_{E'}$.

(1) If E is sufficiently complete (has Property 1), so is E'.

(2) If E' is consistent (has Property 2), so is E.

Lemma 4.1 Let E be a sufficiently complete set of equations and E' be a set of equations containing E. Then the following are equivalent.

(1) E' is consistent.

(2) E is consistent and E' (equivalently each equation in E' - E) holds in the final model Fm(E) of $SP = (S, \sum, E)$.

Proof. (1) => (2): By Proposition 4.1 it suffices to show that E' holds in Fm(E). Let $M = N$ be any equation in E'. Suppose C is a context of sort s_0 in S_0 and θ a ground substitution, then $C(M\theta) \equiv_{E'} C(N\theta)$. Note that both $C(M\theta)$ and $C(N\theta)$ are ground terms of sort s_0. By assumption there are ground terms M_0 and N_0 in $T(\Sigma_0)_{s0}$ such that $C(M\theta) \equiv_E M_0$ and $C(N\theta) \equiv_E$

N_0. Thus, $M_0 \equiv_{E'} N_0$. Since E' is consistent, the above formula implies $M_0 \equiv_{E0} N_0$, therefore $C(M\theta) \equiv_E C(N\theta)$. This means that the equation M = N holds in Fm(E) by Theorem 2.4.

(2) => (1): Let M_0 and N_0 be any terms in $T(\Sigma_0)$ of sort s_0 in S_0 such that $M_0 \equiv_{E'} N_0$. Then there exists a finite sequence of terms in $T(\Sigma)_{s0}$ $M_0 = t_0, \ldots, t_n = N_0$ such that $t_{i-1} \equiv_{E'} t_i$ for i = 1,...,n. By definition for each t_i in the sequence there is an equation $l_i = r_i$ in E', an occurrence u_i and a ground substitution θ_i such that

(a) $t_{i-1}/u_i = l_i\theta_i$, $t_i = t_{i-1}(u_i \leftarrow r_i\theta)$, or
(b) $t_{i-1}/u_i = r_i\theta_i$, $t_i = t_{i-1}(u_i \leftarrow l_i\theta)$.

Let $C_i(x)$ be a context defined by $C_i(x) = t_i(u_i \leftarrow x)$ for each i, where x is a variable which does not appear in the sequence. Since each equation $l_i = r_i$ in E' holds in Fm(E),

$t_{i-1} = C_i(l_i\theta) \equiv_E C_i(r_i\theta) = t_i$, or
$t_{i-1} = C_i(r_i\theta) \equiv_E C_i(l_i\theta) = t_i$,

corresponding to satisfaction of (a) or (b) above respectively for each i. Hence, $M_0 \equiv_E N_0$. Since E is consistent, $M_0 \equiv_{E0} N_0$.

Corollary 4.1 Let E be a consistent and sufficiently complete set of equations and E' a set of equations such that E' contains E. Then, E' holds in the final model Fm(E) of E if and only if E' is consistent.

Recall that Σ_0 is partitioned into Σ_0^c a set of constructors and Σ_0^e a set of extenders. Based on Huet and Hullot's work in (Huet & Hullot 82) we have the subsequent modified results.

Proposition 4.2 Let E be a consistent and sufficiently complete set of equations. For two terms M, N of sort s_0 in S_0, M = N holds in the final model Fm(E) if and only if $M\theta \equiv_E N\theta$ (equivalently $M\theta \equiv_{E0} N\theta$) for every ground substitution θ.

Lemma 4.2 Assume that E is a consistent and sufficiently complete set of equations with property 1 and 2. Let $M = c(M_1,\ldots,M_n)$, $N = c(N_1,\ldots,N_n)$ with c in Σ_0^c. M = N holds in the final model Fm(E) if and only if $M_i = N_i$ holds in Fm(E) for i = 1,...,n.

Proof. To verify the "if part" suppose $M_i = N_i$ holds in Fm(E) for i = 1,...,n. Let C(x) be a context and θ a ground substitution. Then

$$\begin{aligned} C(M\theta) &= C(c(M_1\theta, M_2\theta \ldots , M_n\theta)) \\ &\equiv_E C(c(N_1\theta, M_2\theta \ldots , M_n\theta)) \\ &\quad \ldots \\ &\equiv_E C(c(N_1\theta, N_2\theta \ldots , N_n\theta)) \\ &= C(N\theta). \end{aligned}$$

On the other hand, suppose that the equation M = N holds in Fm(E), $c(M_1\theta,\ldots,M_n\theta) \equiv_E c(N_1\theta, \ldots, N_n\theta)$ for every ground substitution θ by Proposition 4.2. Since $M_i\theta$'s and $N_i\theta$'s are the ground terms of sorts in S_0, by assumption for each i we can choose constructor terms t_i, t_i' in $T(\Sigma_0^c)$ such that $M_i\theta \equiv_E t_i$ and $N_i\theta \equiv_E t_i'$. Thus, $c(t_1,\ldots,t_n) \equiv_E c(t_1',\ldots,t_n')$. This implies $c(t_1,\ldots,t_n) = c(t_1', \ldots,t_n')$. That is $t_i = t_i'$ for each i. From this we can deduce $M_i\theta \equiv_E N_i\theta$. Since both $M_i\theta$ and $N_i\theta$ are terms of sorts in S_0 $M_i = N_i$ holds in Fm(E) for each i by Proposition 4.2.

Corollary 4.2 Let E be a consistent and sufficiently complete set of equations and E' be a set of equations containing E. Suppose M = $c(M_1,\ldots,M_n)$, N = $c(N_1,\ldots,N_n)$ with c in Σ_0^c and E' contains the equation M = N. Let define E" by E" = E' - $\{M = N\}$ U $\{M_i = N_i \mid 1 \leq i \leq n\}$. Then, E' is consistent if and only if E" is consistent. If E' is consistent (equivalently E" is consistent), E" satisfies Property 2, furthermore Fm(E') = Fm(E").(Note that both E' and E" are sufficiently complete and satisfy Property 1).

Lemma 4.3 Let E' be a sufficiently complete set of equations with properties 1 and 2. If $c(M_1,\ldots,M_n) \equiv_{E'} c'(N_1,\ldots,N_n)$ holds with c, c' in Σ_0^c and $c \neq c'$, then E' is not consistent.

Proof. Assume that E' is consistent. Suppose $(M_1,\ldots,M_n) \equiv_{E'} c'(N_1,\ldots,N_n)$ holds, $c(M_1\theta,\ldots,M_n\theta) \equiv_{E'} c'(N_1\theta,\ldots,N_n\theta)$ for every ground substitution θ. Since E' is sufficiently complete and $M_i\theta$'s and $N_i\theta$'s are ground terms of sorts s_i in S_0, there are ground terms p_i, q_i in $T(\Sigma_0)$ such that $M_i\theta \equiv_{E'} p_i$, $N_i\theta \equiv_{E'} q_i$ for i = 1,...,n. Hence, $c(p_1,\ldots,p_n) \equiv_{E'} c'(q_1,\ldots,q_n)$. Consistency of E implies $c(p_1,\ldots,p_n) \equiv_{E0} c'(q_1,\ldots,q_n)$. By property 1 of E there are constructor terms p_i', q_i' such that $p_i \equiv_{E0} p_i'$, $q_i \equiv_{E0} q_i'$ for each i. So that $c(p_1',\ldots,p_n') \equiv_{E0} c'(q_1',\ldots,q_n')$, and $c(p_1',\ldots,p_n') \neq c'(q_1',\ldots,q_n')$. This contradicts the Property 2 of E.

Corollary 4.3 Let E' be a sufficiently complete set of equations with property 1 and 2. Let $M = c(M_1,\ldots,M_n)$ with c in Σ_0^c and $M \equiv_{E'} N$. If $N = c'(N_1,\ldots,N_n)$ with c in Σ_0^c and $c \neq c'$, or N is a variable, then E' is not consistent.

Proof. The former case follows from Lemma 4.3. For the latter case consider a ground substitution θ that substitutes a term $c'(N_1,\ldots,N_n)$ with c' in Σ_0^c, $c \neq c'$ for N. Then this case reduces the former one.

Now we present the modified Knuth-Bendix completion algorithm to prove algebraic laws in the algebra of FP programs. In the following, E_i is a finite set of equations, and R_i is a finite set of rewriting rules for i. Every rewriting rule in R_i has a label, which is a natural number. We denote by k : l -> r the rewriting rule with label k. Finally every rewriting rule in R_i is marked or unmarked.

A Proof Algorithm

Let FP = (S,Σ,E) be the specification of FP system FUNC introduced in chapter 3 and R the set of rewriting rules in FUNC. Note that for the ground object terms M, N, $M \equiv_E N$ iff M! = N!, where M! denotes the unique R-normal form of M.

Initial Data: an equation t = t' to be tested as a law;
a reduction ordering $\succ$

Let $E_0 := \{ t = t' \}$;
$R_0 := R$;
$i := 0$; $p := |R|$.

Assume that every rule in R has a unique label and marked.

loop
while $E_i \neq \emptyset$ do
Begin
(A) Reduce Equation:
Select an equation M = N in E_i; delete it from E_i:
$E_i := E_i - \{ M = N \}$.
Reduce the terms M (resp. N) to obtain the R_i-normal form M!

(resp. N!) by applying rules in R_i of any order, until none applies. (This can be guaranteed since R_i is not only confluent, but Noetherian.)

if M! = N!, then let

$E_{i+1} := E_i$; $R_{i+1} := R_i$; $i := i+1$.

elseif both M! and N! are object terms such that at least one of them begins with a constructor (if M! is a object term, so is N!, and vice versa.),

if $M! = c(M_1,\ldots,M_n)$ with c in Σ_0^c,

(a) if $N! = c(N_1,\ldots,N_n)$, then let

$E_{i+1} := E_i \cup \{ M_j = N_j \mid 1 \leq j \leq n \}$;

$R_{i+1} := R_i$; $i := i+1$.

(b) if $N! = c'(N_1,\ldots,N_n)$ with c' in Σ_0^c, $c' \neq c$, or N! = x (N! is a variable), then stop with "disproof".

(c) if $N! \succ M!$, then let

$l = N!$, $r = M!$.

(d) otherwise stop with "failure".

else do as above.

elseif $M! \succ N!$, then let

$l = M!$, $r = N!$.

elseif $N! \succ M!$, then let

$l = N!$, $r = M!$.

else stop with "failure".

(B) Add New Rules:

Let K be the set of labels k of rules $l_k \to r_k$ in R_i whose left-hand side l_k is reducible by $l \to r$, say to l_k'. Let

$E_{i+1} := E_i \cup \{ l_k' = r_k \mid k : l_k \to r_k \text{ in } R_i \text{ with } k \text{ in } K \}$;

$p := p+1$;

$R_{i+1} := \{ j : l_j \to r_k' \text{ in } R_i \text{ with } j \notin K \} \cup \{ p : l \to r_j' \}$,

where r_j' is a normal form of r_j using rules from $R_i \cup \{ l \to r \}$. The rule coming from R_i are marked or unmarked as they were in R_i, the new rule $l \to r$ is unmarked;

$i := i+1$.

end.

endwhile

(C) Compute Critical Pairs:

If all rules in R_i are marked, stop with success.

Otherwise, select an unmarked rule in R_i with the least label k. Let

E_{i+1} be the set of all critical pairs computed between rule k and any rule of R_i of label not greater than k. Let R_{i+1} be the same as R_i, except that rule k is marked;

i := i+1.

endloop

Before giving the results of the proof algorithm, we state it more precisely. The algorithm goes through successive passes on sets of equations and rewriting rules. Initially, we set $R_0 = R$, and $E_0 = \{ t = t' \}$ the set consisting of only one equation to be tested as a law. By construction every R_i is a Noetherian term rewriting system, because $l > r$ for every rule $l \dashrightarrow r$ in R_i. In the main loop of the algorithm an equation is removed from E_i and simplified by the current simplification set R_i into a candidate pair (M!, N!). In the standard completion algorithm if M! and N! are not comparable in the ordering $>$, the algorithm stops with failure. Otherwise, let l be the greatest, and r be the smallest. We place the new rewriting rule $l \dashrightarrow r$ in R_{i+1}. The rules from R_i whose left-hand side is simplifiable by the new rules are placed in E_{i+1}, together with the remaining equations. The other rules from R_i are placed in R_{i+1}, after possible simplification of their right-hand side, using rules in $R_i \cup \{ l \dashrightarrow r \}$.

In our proof algorithm, new failure cases are added, when the ordering would oblige to consider a rewriting rule with a constructor in $\sum_0^c$ as a leading operation symbol at its left-hand side. Also we have a new case of termination, whenever M! and N! start with different constructors, or one starts with a constructor and the other is a single variable; we then stop with "disproof" as a result.

Besides the cases of failure and disproof, the only departure from the standard completion algorithm is when $M! = c(M_1,\ldots,M_n)$, $N! = c(N_1,\ldots,N_n)$ for some constructor c in $\sum_0^c$. We then set R_{i+1} to R_i, and E_{i+1} to $\{ M_i = N_i \mid 1 \leq i \leq n \}$ added to the remaining equations in E_i. This step is called "induction step" by Huet and Hullot in (Huet & Hullot 82). To state a relationship between E_i, R_i and E_{i+1}, R_{i+1} more precisely let

$$\mathcal{E}_i = E_i \cup R_i.$$

By Huet's result we have in general $\equiv_{\mathcal{E}i} \subset \equiv_{\mathcal{E}i+1}$. If the step from i to i+1 is not induction step $\equiv_{\mathcal{E}i} = \equiv_{\mathcal{E}i+1}$.

At every step i, whenever $\mathcal{E}_i$ is sufficiently complete (satisfies property 1), so is $\mathcal{E}_{i+1}$ (so does $\mathcal{E}_{i+1}$) by Proposition 4.1. And initially $\mathcal{E}_0 = \{ t = t' \} \cup$

R is sufficiently complete and satisfies property 1, we can deduce that every $\mathfrak{L}_i$ is sufficiently complete and satisfies property 1, where we treat R as a set of equations rather than a set of rewriting rules.

Proposition 4.3 Let E be a set of equations defining a canonical term rewriting system, i.e. confluent and Noetherian, such that there is no rule whose left-hand side is a constructor term. Then E satisfies the property 2.

Lemma 4.4 If $\mathfrak{L}_i$ is consistent (satisfies Property 2), so is $\mathfrak{L}_{i+1}$ (does), and furthermore $Fm(\mathfrak{L}_{i+1}) = Fm(\mathfrak{L}_i)$, i.e., the final model of $\mathfrak{L}_{i+1}$ coincides with the one of $\mathfrak{L}_i$.

Proof. If step i is as in the standard completion algorithm, then $\equiv_{\mathfrak{L}i} = \equiv_{\mathfrak{L}i+1}$, and the result follows. If step i is an induction step, we get the result from Corollary 4.2.

Corollary 4.4 If $E \cup \{ t = t' \}$ is consistent, then $\mathfrak{L}_i$ are consistent, and $Fm(E) = Fm(\mathfrak{L}_i)$.

Theorem 4.1 Main Theorem for the proof procedure in the FP Algebra.

(1) If the algorithm stops with success, the given equation t = t' holds as an algebraic law.

(2) If the algorithm stops with disproof, t = t' does not hold.

(3) If t = t' does not hold, the algorithm stops with either disproof or failure.

Proof. For (1): If the algorithm stops with success, say at step n, the resulting term rewriting system R_n is confluent and Noetherian and contains $E' = E \cup \{ t = t' \}$. By assumption and Proposition 4.3 R_n is sufficiently complete and satisfies property 1 and 2, since there is no rule whose left-hand side is a constructor term. So that the same result holds for E'. Now we shall show that R_n is consistent. Suppose it is not, then there are two ground terms M, N in $T(\Sigma_0)$ such that $M \equiv_E N$, $M \not\equiv_{E0} N$. Since E satisfies property 2, we can choose constructor terms M_0 and N_0 such that $M_0 \equiv_{E0} M_0$, $N_0 \equiv_{E0} N_0$. Hence, $M_0 \equiv_E N_0$, $M_0 \neq N_0$. However, M_0, N_0 are R_n-normal forms and $M_0 \equiv_E N_0$ implies $M_0 = N_0$. This contradicts that $M_0 \neq N_0$. Thus E' is consistent. By Lemma 4.4 t = t' holds in the final model of the specification. We also get from Corollary 4.4 that $Fm(E) = Fm(R_n)$.

For (2): If the algorithm stops with disproof at step n, it means that $\mathcal{E}_n$ is not consistent according to Corollary 4.4. By Lemma 4.4 this means that E' is not consistent. Lemma 4.1 shows that t = t' does not hold in Fm.

For (3): Assume that the equation t = t' does not hold in Fm. By Lemma 4.1 E' is not consistent. So without loss of generality let M, N be constructor terms such that $M \equiv_{E'} N$, and $M \not\equiv_{E0} N$ (i.e., $M \neq N$). If the algorithm doesn't terminate, it generates an infinite confluent set of rewriting rules R', and M and N are eventually reducible to exactly the same term. But this is impossible since no rules generated may reduce a constructor term. Hence M = N. But this contradict the assumption. So that the algorithm stops either with disproof or failure.

5. CONCLUSION

In this paper we have presented a proof procedure for algebraic laws in the algebra of Backus' FP programs. At first we have formulated the Backus' FP system as a term rewriting system, some properties of the resulting system and its relationship with the final model of the associated specification were investigated. Finally we presented a proof procedure for the algebraic laws in the algebra of FP programs.

ACKNOWLEDGMENT

We would like to thank Mr. Glenn Mansfield for his looking over this technical memo and valuable comments.

REFERENCES

(Backus 78) Backus, J.H.
Can programming be liberated from the Von Neumann style ? - A functional style and its algebra of programs, Comm. ACM, Vol.20, No.8 (1978) pp.613-641.

(Backus 81a) Backus, J.H.
Function level programs as mathematical objects, ACM Conf. on Functional Programming Language and Computer Architecture (1981)

pp.1-10.

(Backus 81b) Backus, J.H.
The algebra of functional programs, functional level reasoning, linear equations and extended definitions, Springer Lecture Notes in Computer Science, Vol.107 (1981) pp.1-43.

(ADJ 78) Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B.
An initial algebra approach to the specification, correctness, and implementation of abstract data types, in Current Trends in Programming Methodology, Vol.4, ed. Yeh,R., Prentice-Hall (1978) pp.80-149.

(Huet 80) Huet, G.
Confluent reductions: Abstract properties and application to term rewriting systems, J.ACM, Vol.27. No.4 (1980) pp.797-821.

(Huet 81) Huet, G.
A complete proof of correctness of Knuth-Bendix completion algorithm, JCSS, Vol.23 (1981) pp.11-21.

(Huet & Hullot 82) Huet, G. and Hullot, J.M.
Proof by induction in equational theories with constructors, JCSS, Vol.25 (1982) pp.239-265.

(Huet & Oppen 80) Huet, G. and Oppen, D.C.
Equations and rewrite rules, A survey, in Formal Language Theory, Academic Press (1980) pp.349-405.

(Möller 84) Möller, B.
An algebraic semantics for busy and lazy evaluation and its application to functional languages, Springer Lecture Notes in Computer Science ICALP 83, Automata Language and Programming (1984) pp.513-526.

(Dosch & Möller 84) Dosch, W. and Möller, B.
Busy and lazy FP with infinite objects, ACM Conf. on Lisp and Functional Programming (1984) pp.282-292.

(Musser 78) Musser, D.R.
A data type verification system based on rewrite rules, 6th Texas Conf.

on Computer Science (1978).

(Musser 80) Musser, D.R.
On proving inductive properties of abstract data types, 7th ACM Symp. on Principles of Programming Language (1980).

(Williams 81) Williams, J.H.
On the development of the algebra of functional programming, Springer Lecture Notes in Computer Science Vol.107 (1981) pp.733-757.

(Wand 79) Wand, M.
Final algebra semantics and data type extensions, JCSS, Vol.19 No.1 (1979).

分散計算モデルと通信計算量

正田

1985年 3月

分散計算 distributed computingとは、ネットワークによって結ばれた複数個のプロセサが、その間でメッセージの交換を行いながら、協同して全体として一つの計算を行うというものである。これはプログラミング言語としては C.A.R. Hoare による CSP (Communicating Sequential Processes) と対応している。最近のマルチプロセサシステムや計算機ネットワークの発展に応じて、分散計算のモデルの理論的研究が盛んになっている。1982年以来毎夏、ACM の SIGACT と SIGOSの主催で、Symposium on Principles of Distributed Computingというシンポジウムが開催されている。

分散計算モデルにおける理論的な研究テーマには、

・プログラムの検証などの公理的取り扱い

・fault tolerancy （たとえばビザンチン将軍問題）

・デッドロックの発見、fairnessなど

・応用における種々の具体的なアルゴリズムの考案とその計算量の解析

・確率的なアルゴリズム、その検証

などがあり、いずれも興味深い。

小文の目的は二つあり、この計算モデルの仮定を述べることと、このモデルにおける各種のアルゴリズムの通信計算量 communication complexity というテーマについて、文献(7) を主に参考にしながら最近の話題を少々紹介することである。通信計算量とは計算過程において交換されたメッセージの総数で、この計算モデルにおける計算量の概念を捉える。モデルの仮定の細部においていろいろの面白い変形がありえて、その変形によって計算のパワーつまり計算量の上限／下限にはっきり差が生じる。

1．分散計算モデル

分散計算のモデルの基本的な仮定は次のようなものである。

(i) 一つの無向グラフの頂点（ノード）がプロセサ、辺がプロセサ間の通信リンクを表す。

(ii) 各プロセサはリンクを通じてメッセージをすぐ隣のプロセサに送ることができる。

(iii) 計算はいくつかのソースノードから開始され、計算終了後、結果はいくつかのシンクノードにおいて得られる。

(iv) 各プロセサは同一のアルゴリズムを遂行する。

以上だけでは細部にはっきりしない点が残っていて、いろいろの変形がありえる。４節でいくつかを述べる。

２．通信計算量

分散計算の遂行において主要な部分はメッセージの交換（send/receive）であると見なされる。それゆえ、分散計算アルゴリズムの計算コストを考えるときは、各プロセサ内における局所的な計算時間は無視できるものとし、全体としての計算量は、計算過程において通信リンク上で交換されたメッセージの総数とする。これをそのアルゴリズムの通信計算量 communication complexity という。

３．応用における具体的な問題の例

グラフ理論における問題をはじめとして、いろいろの具体的な分散アルゴリズムの通信計算量が研究されている。次の選出問題 Election, Extrema-finding は基本となるものと見なされ、かなりよく調べられている。

「各プロセサに一つの整数値が与えられている。これらのうちで最大のものをもつプロセサを見つけよ」

この他に、minimum spanning tree や最短距離問題などが扱われているが、扱われる範囲はこれから広がりそうである。

４．モデルの仮定の細部　－　計算量に影響を及す因子

分散計算のモデルの仮定は細かい点でいろいろの変形ないし選択がありえて、それがアルゴリズムに影響を与える。たとえば次のような変形の因子がある。

（Ａ）リンクが一方向か／双方向か　(uni/bi-directional)
　　メッセージが一つの通信リンクの上を一方向のみに送られるか／双方向に送られうるか

（Ｂ）ネットワークのグラフのトポロジー　(topology)
　　リングか、完全グラフか、一般のグラフか。

この（Ａ）と（Ｂ）の因子それぞれが、計算のパワーに（通信計算量に）影響するであろうということは明らかである。しかしこれら以外にも、次のような因子があることが最近わかってきた。

（C）方向の知覚　（global sense of direction ）

（D）トポロジーの自覚　（awareness of topology ）

この二つの因子が面白い。

5．方向の知覚

方向の知覚とは、たとえば環（リング）状のネットワークにおいて、各ノードにおける「左」と「右」の方向が、全頂点において同調しているかどうかということである。この方向の知覚があるかないかによって、一つの問題の分散計算のむずかしさが違ってくるであろうことは、リングの場合には容易に想像がつくであろう。

ネットワークが完全グラフをなす場合には、方向の知覚とは（やや人為的だが）ハミルトン閉路（各頂点をちょうど１回だけ通る閉路）が一つ定められていてその知識を利用することとする。完全グラフにおける前記の選出問題は、方向の知覚があるかどうかによって、計算量のオーダは異なる：

方向の知覚があるとき	O(n)	で可
方向の知覚がないとき	Ω(n log n)	が下限

6．トポロジーの自覚

あるアルゴリズムにとってトポロジーの自覚があるかないかとは、そのネットワークの全体的なトポロジー（リング、完全グラフ、その他の別）の知識を使用するかということではなく（もっと強く）、グラフの局所的な構造つまり adjacency matrix の知識を使用するかどうかということである。

これも通信計算量に差をもたらす。たとえば、完全グラフにおける選出問題では、

トポロジーの自覚があるとき	Ω(n log n)	が下限
トポロジーの自覚がないとき	Ω(n*n)	が下限

とはっきり差のあることが最近わかっている。

●参考文献

選出問題を中心に文献を少しあげる。通信計算量についてはくわしい文献表が〔7〕にある。

〔1〕 E. Chang and R. Roberts : An improved algorithm for decentralized extrema-finding in circular configurations of processes, Comm. ACM, 22 (1979), 281-283.

(2) D. Dolev, M. Klawe and M. Rodeh : An O(n log n) undirectional distributed algorithm for extrema finding in a circle, J. Algorithms, 3 (1982), 245-260.

(3) E. Gafni and Y. Afek : Election and traversal in unidirectional networks, Proc. Third Annual ACM Symp. on Principles of Distributed Computing, 1984, pp. 190-198.

(4) E. Korach, S. Moran and S. Zaks : Tight lower and upper bounds for some distributed alogorithms for a complete network of processors, Proc. Third Annual ACM Symp. on Principles of Distributed Computing, 1984, pp. 199-207.

(5) G. L. Peterson : An O(n log n) unidirectional algorithm for the circular extrema problem, ACM TOPLAS, 4 (1982), 758-762.

(6) D. Rotem, E. Korach and N. Santoro : Analysis of a distributed algorithm for extrema finding in a ring, School of Computer Science, Carleton Univ., SCS-TR-61, August 1984.

(7) N. Santoro : Sense of direction, topological awareness and communication complexity, ACM SIGACT News, 16(2), Summer 1984, pp. 50-56.

(8) J.-R. Sack, N. Santoro and J. Urrutia : O(n) election algorithms in complete networks with global sense of orientation, School of Computer Science, Carleton Univ., SCS-TR-49, May 1984.

プロセッサ・メモリ結合網のあるクラスの性能評価

計算機ハードウェアの性能が向上するにつれて、時間計算量の高い人工知能分野や画像、音声などの情報量が多い対象の処理への計算機応用の要求が高まって来ている。そして処理速度向上のため、様々な角度からの並列計算機アーキテクチャ（array-processor,dataflow-machine,Prolog-machine など）の研究が盛んに行われている。これらの研究で本質的に解決しなければならない難問の一つは、プロセッサ群とメモリ群との間（あるいは、プロセッサとプロセッサ間）の効率のよい結合網の開発である。例えば、「ｎ個のプロセッサがｎ個のメモリ・バンクを共有しており、それぞれアクセスしたいメモリ・バンクを指定したとき、効率よく全てのアクセス要求を実現する結合網とそのルーティング・アルゴリズムはどのようなものか？」という問題は、効率のよい並列計算機アーキテクチャを開発するうえで、基本的な問題である。この問題はグラフ理論の問題として抽象化されるが、結合網の転送時間、ハードウェア量、ルーティング・アルゴリズムなどの研究がなされるべきである。ここでは、制限されたプロセッサ・メモリ結合機能を抽象化したpermuterを中心に、superconcentrator, generalizer, 及びdistribution networkの性能比較を行なう。

Ｈ．１．定義

一般的なグラフ理論の用語は既知のものとする。

Ｇ＝（Ｖ，Ｅ）をサイクルのない有向グラフとする。

３つ組　Ｎ＝（Ｇ，Ａ，Ｂ）　をｎ－ｎｅｔｗｏｒｋと呼ぶ（但し、Ａ，Ｂ⊂Ｖ，Ａ∩Ｂ＝φ，｜Ａ｜＝｜Ｂ｜＝ｎ）。また、Ａをｉｎｐｕｔ、Ｂをｏｕｔｐｕｔと呼ぶ。／＊Ｎの物理的解釈の仕方　：　①Ａをプロセッサ群、Ｂをメモリ群とするプロセッサ・メモリ間結合網を表わす。②Ａはプロセッサ群、かつＡ＝Ｂで、プロセッサ間結合網を表わす。＊／

Ｇの辺の数｜Ｅ｜をｓｉｚｅ（Ｎ）と書く。

あるｉｎｐｕｔからあるｏｕｔｐｕｔへの最長パスの長さをｄｅｐｔｈ（Ｎ）と書く。／＊　ｄｅｐｔｈ（Ｎ）は、プロセッサ・メモリ間（あるいは、プロセッサ間）結合網Ｎでのデータ転送時間と解釈してよく、Ｎの時間性能を示す重要な尺度である。＊／

有向グラフＧ＝（Ｖ，Ｅ）の２つの部分グラフＧ１＝（Ｖ１，Ｅ１）とＧ２＝（Ｖ２，Ｅ２）が、Ｖ１∩Ｖ２＝φのとき、Ｇ１とＧ２は頂点独立であると言う。２つ以上の部分グラフに対しても、頂点独立は同様に定義される。

ｎ－ｎｅｔｗｏｒｋ　Ｎ＝（Ｇ，Ａ，Ｂ）で、Ａの任意の部分集合Ａ’と、Ｂの任意の部分集合Ｂ’（但し、｜Ａ’｜＝｜Ｂ’｜）に対して、Ａ’からＢ’への｜Ａ’｜本の頂点独立なパスが存在するものをｎ－ｓｕｐｅｒｃｏｎｃｅｎｔｒａｔｏｒと言う。

ｎ－ｎｅｔｗｏｒｋ　Ｎ＝（Ｇ，Ａ，Ｂ）で、任意の関数　ｈ：Ａ→Ｉ（但し、Ｉは非負整数の集合で、Σ｛ｈ（ａ）｜ａ∈Ａ｝≦ｎ）が与えられたとき、各ａ∈Ａに関してａを根とし、ｈ（ａ）個のｏｕｔｐｕｔを葉とする木Ｔ（ａ）がとれて、木の集合｛Ｔ（ａ）｜ａ∈Ａ｝を頂点独立とできるものをｎ－ｇｅｎｅｒａｌｉｚｅｒと言う。

ｎ－ｎｅｔｗｏｒｋ　Ｎ＝（Ｇ，Ａ，Ｂ）で、任意の全単射写像　π：Ａ→Ｂ　が与えられたとき、各ａ∈Ａに関してａからπ（ａ）へのパス　Ｐ（ａ）がとれて、パスの集合｛Ｐ（ａ）｜ａ∈Ａ｝を頂点独立とできるものをｎ－ｐｅｒｍｕｔｅｒ（又は、ｎ－ｃｏｎｎｅｃｔｏｒ，ｎ－ｐｅｒｍｕｔａｔｉｏｎｎｅｔｗｏｒｋ）と言う。

ｎ－ｎｅｔｗｏｒｋ　Ｎ＝（Ｇ，Ａ，Ｂ）で、任意の関数　ｈ：Ａ→Ｂ（但し、Ｂは、集合Ｂの部分集合全ての集合で、Ａの相異なる２頂点ａ、ａ'に関して、ｈ（ａ）∩ｈ（ａ'）＝φとなる）が与えられたとき、各ａ∈Ａに関してａを根としｈ（ａ）のｏｕｔｐｕｔを葉とする木Ｔ（ａ）がとれて、木の集合｛Ｔ（ａ）｜ａ∈Ａ｝を頂点独立とできるものをｄｉｓｔｒｉｂｕｔｉｏｎ　ｎ－ｎｅｔｗｏｒｋ（又は、ｇｅｎｅｒａｌｉｚｅｄ　ｎ－ｃｏｎｎｅｃｔｏｒ）と言う。

ｎ－ｎｅｔｗｏｒｋ　Ｎ＝（Ｇ，Ａ，Ｂ）に於いて、ｉｎｐｕｔからｏｕｔｐｕｔへの全てのパスが同じ長さのとき、Ｎはｓｙｎｃｈｒｏｎｏｕｓであると言う。

Ⅱ．２．各種ｎ－ｎｅｔｗｏｒｋのｓｉｚｅ

ここでは、n-permuter，n-generalizer，n-superconcentrator　及びdistribution n-network のｓｉｚｅに関する上界と下界を表の形でまとめる。

［記法］　s(n-superconcentrator) に関して

$s(n)$　：n-superconcentrator の最小ｓｉｚｅ

$s_k(n)$：深さが高々ｋのn-superconcentrator の最小ｓｉｚｅ

$s^*(n)$：synchronous なn-superconcentrator の最小ｓｉｚｅ

$s^*_k(n)$：深さがｋのsynchronous なn-superconcentrator の最小ｓｉｚｅ

また、ｇ（n-generalizer）、ｐ（n-permuter）、ｄ（distribution　n-network）に関しても同様の記法を用いる。

［定義］関数λとβ

$$\begin{cases} A(0,j)=2^j & \cdots \quad j\geqq 1 \\ A(i,1)=2 & \cdots \quad i\geqq 1 \\ A(i,j)=A(i-1,A(i,j-1)) & \cdots \quad i\geqq 1,\ j\geqq 2 \end{cases}$$

各ｉに対して、

$\lambda(i,x)=\min\{j \mid A(i,j)\geqq x\}$

／＊　λ（ｉ，ｘ）は、log ｘのｉ乗の近似と考えてよい。＊／

$\beta(x)=\min\{i \mid A(i,j)\geqq x\}$

即ち、$\beta(x)=\min\{i \mid \lambda(i,x)\leqq i\}$

／＊β（ｘ）は、任意の原始帰納関数の逆関数よりゆっくり増加する関数＊／

n－p e r m u t e r

	下　界	深さkの制限	上　界
P(n)	$\geqq 6n\log_6 n + \theta(n)$ [8]		
		ある $k=2\log_2 n + \theta(\log\log n)$	$\leqq 6n\log_2 n + O(n(\log n)^{1/2})$ [16]
$P_k(n)$	$\geqq k\, n^{1+1/k}$ [18]	$k=2k'-1$ （各k'）	$\leqq 2k'(1/2)^{1/k'} n^{1+1/k'} + O(n)$ [16]
			$O(n^{1+1/k}(\log n)^{1/k})$ [18]

n－g e n e r a l i z e r

	制　限	下　界	上　界
$g(n)$	ある $k=\theta(\log^2 n)$ ある $k=\theta(n^{\alpha})$ $0<\alpha<1$		$\leqq 120n+O(\log^2 n)$ [15] $\leqq 79.7n+O(n^{\alpha})$ [2]
$g_k(n)$	$k=2$ $k=2k'$ （各$k'\geqq 2$） $k=2k'-1$ $n=d1\cdot d2\cdots dk'$ （各$di\geqq 2$） $n=2^{k-1}m$ $n=d1\cdot d2\cdots dk$ （各$di\geqq 2$）		$O(n\log^3 n)$ [5] $O(n(\lambda(k',n)^2)$ [5] $\leqq (d1+2d2+\cdots+2dk')n$ [19]† $\leqq (2(k-1)+m)n$ [4] † $\leqq (d1+\cdots+dk)n$ [5] †
$g_k^*(n)$	$k=2k'-1$ （各$k'\geqq 1$）	$\Omega(n\lambda(k',n))$ [5]	

† simple explicit construction

n－superconcentrator

	下　界	深さkの制限	上　界
s(n)	≧5n+ $\theta(\log n)$ [7]		≦238n [20]
		ある　$k=\theta(n^{\alpha})$ $0<\alpha<1$	≦238n [20]
		ある　$k=\theta(\log n)$	≦39n+0(log n) [14] ≦36n+0(log n) [2]
		ある$k=2\beta(n)+\theta(\log\beta(n))$	≦36n+0(n/β(n))[5]
$s_k(n)$	Ω(nlogn) [17]	k=2	0(nlogn) [17] ≦135n($\lambda(1,n)$)[5]
		k=2k'	≦5000n $\lambda(k',n)$[5]
$s_k^*(n)$			≦k·$s_k(n)$ [5]

distribution n-network

	下　界	深さkの制限	上　界
$d(n)$	$\geqq P(n)$ $\geqq 6n\log_2 n$ $+ O(n)$ [8]		$\leqq g(n) + P(n)$
		$k=3k'-2$ $k'=\log_3 n$ $+\theta(\log(\log n))$ となるあるk'に対し	$\leqq 9n\log_3 n$ $+O(n(\log n)^{1/2})$[5]
		$k=5k\log n$	$10n\log n$[12]
		$k=\theta(n\log n)$	$7.6n\log n$ [13]
		$k=3.8\log n$	$7.6n\log n$ [19] ($n^{1-\frac{1}{r}}$ PE のcube かSEで$O(r\log^2 n)^{\dagger}$)
		$k=\theta(\log^2 n)$	$3.8n\log n+120n$[15]
		$k=\theta(\log n)$	$5.8n\log n$ [4] シリアル$O(n\log n)^{\dagger}$
		$k=\theta(m\log n)$ $1 \leqq m \leqq \log n$	$O(kn^{1+\theta(\frac{\log m}{k})})$[11] ($n^{1+\theta(\frac{k}{\log m})}$のcubeか SEで$O(m\log n)^{\dagger}$)
$d_k(n)$	$\geqq P_k(n)$ $\geqq kn^{1+k}$ [18]	$k=k1+k2$ $k=3k'-2$ （各k'に対し）	$\leqq g_{k1}(n)+P_{k2}(n)$
		各$k>4$	$\leqq 3k'(4/9)^{1/k'}n^{1+1/k'}$ $+O(n)$[5]
		$k=3$	$O((n\log n)^{1-1/k})$[5] $O(n^{5/3})$ [12.14]

† スイッチのセットアップ・アルゴリズム

H．3．セットアップ・アルゴリズム

上記の各種ネットワークで、指定されたアクセス・パスを確保するために、各スイッチの状態を決定することを、セットアップと言う。いくらネットワークの転送速度が速くとも、セットアップに要する時間が長ければ、全体として効率のよいネットワークとは言えない。そのために、最近はパラレルなセットアップ・アルゴリズムも研究されている。さらに、ネットワークにとっては理想的なことであるが、特別なセットアップの必要のないセルフ・ルーティング（データの転送先の情報と各スイッチのローカルな情報だけで、各スイッチが独自にルーティングの経路を決定していく）と言う方法も研究されている。

Benes network（ある n-permuter)のセットアップ・アルゴリズム

時間計算量	プロッセサ数	備　　考
O(n log n)[21] O(log ^{2} n)[9] O($n^{1/2}$)[9] O(m log n)[9]	1 n $n^{1/2} \times n^{1/2}$ $n^{1+\frac{1}{m}}$ 各m (1≦m ≦n)	Completely Interconnected Computerで Mesh Connected Computer で Cube Connected Computer 又は Perfect Shuffle Computerで
O(log n)[10]	O	セルフ・ルーティング 但し、できない順列もある

（注）　Benes network は、サイズ nlog n - n/2,深さ 2log n - 1

Benes network 以外のn-permuter

サイズ	深さ	備　考
	O($n^{1/2}$)?	セット・アップは簡単[6]
O(nlog^2 n) O(n m(1+log n - log m) •log n / log m) 各m(2 ≦m ≦n)	O(log^2 n) O(log ^{2} n/logm)	セルフ・ルーティング[3] セット・アップは簡単 O(log^2 n /log m)　[11]

multi-stage n-permuterのset-upアルゴリズム

各 i (1≦i ≦t)stage と(2t+1-(i-1))stage は、1-stage-di-permuter 、(t+1)stageは、1-stage-m-permuterである。（但し、n ≦ d1･d2･･･dt･m ）
1-stage-k-permuterは、(k,k) の完全２部グラフ

2t+1 stage	diの制限	時間計算量
t=1	[d=2]	[O(n) シリアル] [21]
	d は２の巾	O(n)シリアル [1]
	各diは２の巾 (1≦i ≦t)	O(nlogn)シリアル [8] O(log^2 n)パラレル$^+$ [8]
	各diは任意 (1≦i ≦t)	O(n log^2 n)シリアル [8] O(nlog^2 n)パラレル$^+$ [8]

$^+$ 同一のプログラムを実行するn個の同期したプロセッサで計算する。共通メモリへのアクセスで、２つ以上のプロセッサが同一語をアクセスすることはない。

H．4．問題点

以上述べた結合網の抽象化は静的であり、実際の並列計算機アーキテクチャに応用するには、次のような問題点がある。実状には弱すぎる点として、
(i)　プロセッサのアクセス要求の全てが同期して出されるのを前提としている、
(ii)　耐故障性が考慮されていない、
(iii) VLSI化に対する考慮がされていないことが、
また強すぎる点として、
(iv)　同時にすべてのアクセス要求を満たさなければならないこと（定数個程度のプロセッサを定数時間程度待たせてもよい）
などが挙げられる。これらの実際的条件を考慮した結合網及びルーティング・アルゴリズムの研究は今後の課題である。

H．5．ネットワーク、パラレル・アルゴリズム、ＶＬＳＩの将来研究

ネットワーク、パラレル・アルゴリズム、ＶＬＳＩをキーワードとした将来研究について、次のようなものが考えられる。

ネットワークーパラレル・アルゴリズム

☆通信複雑度（ネットワークと問題が与えられて…）
（→疋田先生のレポート参照）
単なるメッセージ量だけを問題にするのではなくて、計算時間とも関連をもたしたメッセージ量の解析

☆プロセッサ配分問題
計算に関するデータの依存関係が与えられたとき、通信量が少なくてすむようなプロセッサの配分は？

☆ルーティング・アルゴリズム（並列アルゴリズムの基本問題）

☆パラレル処理に於いて、あるプロセッサが故障したときに、情深い近所のプロセッサたちが負荷分担して、全体では支障なく処理されるようなパラレル・アルゴリズム？

☆故障に強いネットワーク？
ルーティング・アルゴリズム？

☆ネットワークを多重にして、時間効率を上げれるようなネットワークなんてあるのだろうか？

ＶＬＳＩーネットワーク

☆多種のグラフを埋め込めるような台となるグラフの開発

☆グラフ族の規則的なレイアウト法

ＷＳＩ（Wafer Scale Integration）時代に

☆少種類のセルを多数作って、複雑なネットワークを構成するのに適したグラフ

ＶＬＳＩに向いたグラフの埋め込み
グラフを埋め込んだときに、

☆面積が小さくなるような埋め込み

☆長くなるワイヤがないような埋め込み

☆ワイヤ遅延が小さくなるような埋め込み

☆交差するワイヤの個数が小さくなるような埋め込み

パラレル・アルゴリズムーＶＬＳＩ

安浦先生のレポート参照

文　献

[1] A.Andresen.
"The Looping Algorithm Extended to Base 2 Rearrangeable Switching Networks".
IEEE Trans. on Comp., C-25, pp.1057-1063, 1977.

[2] L.A.Bassalygo.
"Asymptoticcally Optimal Switching Circuits".
Prob. Info. trans., 17, pp.206-211.1981.

[3] K.E.Batcher.
"Sorting networks and their applications".
Proc. AFIPS 1968 SJCC Vol.32, Montrale, NJ, AFIPS Press pp.307-314.

[4] K.M.Chung and C.K.Wong.
"Construcation of a Generalized Connector with 5.8n log n Edges".
IEEE Trans. on Comp., 29, pp.1029-1032.1980.

[5] D.Dolev, C.Dwork, N.Pippenger and A.Wigderson.
"SUPERCONCENTRATORS,GENERALIZERS AND GENERALIZED CONNECTORS WITH LIMITED DEPTH".
Proc. of the 15th Ann. ACM Symp. on Theory of Computing, Boston, Massachusetts, April 25-27, 1983.

[6] T.Lang and H.stone.
"A Shuffle-exchange network with simplified control".
IEEE Trans. on Comp., C-25, pp.55-65, Jan. 1976.

[7] G.Lev.
"Size Bounds and Parallel Algorithms for Networks".
Thesis, Department of Computer Science, University of Edinburgh, Septemmber 1980.

[8] G.Lev, N.Pippenger and L.Valiant.
"A Fast Parallel Algorithm for Routing in Permutation Networks".
IEEE Trans. on Comp. C-30, 2, pp.93-100, Feb. 1981.

[9] D.Nassimi and S.Sahni.
"Parallel algorithms to set-up the Benes permutation network".
Proc. Workshop on Intersonnection Networks for Parallel and Distributed Processing, Purdue Univ., Lafayette, IN, April 1980.

[10] D.Nassimi and S.Sahni.
"A Self-Routing Benes Network and Parallel Parmutation Algorithms".
IEEE Trans. on Comp., C-30, 5, pp.332-1340, May, 1981.

[11] D.Nassimi and S.Sahni,
"Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network",
J. ACM, 29 ,pp.642-667,1982.
[12] Yu.P.Ofman,
"A Universal Automaton",
Trans. Moscow Math. Soc., 14, pp.200-215,1965.
[13] N.Pippenger,
"The complexity theory of switching networks",
MIT Res. Lab. of Electronics, Rep. TR-487, 1973.
[14] N.Pippenger,
"Superconcentrators",
SIAM J. Comp., 6, pp.298-304,1977.
[15] N.Pippenger,
"Generalized Connectors",
SIAM J. Comp., 7 , pp.510-514,1978.
[16] N.Pippenger,
"On Rearrangeable and Non-Blocking Switching Netwworks",
J. Comp. and Sys. Sci. 17, pp.145-162, 1978.
[17] N.Pippenger,
"Superconcentrators of Depth 2",
J. Comp. and Sys. Sci., 24, pp.82-90,1982.
[18] N.Pippenger and A.C.-C. Yao,
"Rearrangeable Networks with Limited Depth",
SIAM J. Alg. and Disc. Meth.,3, 4, pp.411-417, 1982.
[19] C.D.Thompson,
"Generalized Connection Networks for Parallel Processor Interconnection",
IEEE Trans. on Comp., 27, pp.1119-1125,1978.
[20] L.G.Valiant,
"Graph-Theoretic Properties in Computational Complexity",
J. Comp. and Sys. Sci., 13, pp.278-285,1976.
[21] A.Waksman,
"A permutation network network",
JACM, 15, 1, pp.159-163, 1968.

Read-Only Annotationの Unificationとその応用

枚田正宏

はじめに

ユニフィケーションをもとにした並列処理言語 Concurrent Prologについては，その意味論および実現方法をめぐる批判的な検討が行なわれて来た．特に，Read-Only Annotationのユニフィケーションについては，Shapiro による処理系の不都合な点の指摘が相次いだ．このメモでは，変数X とそのRead-Only Annotation X? とのユニフィケーションの結果に関して一意性をみたす算法を与える．

１．木及びグラフのユニフィケーション

本来，ユニフィケーションは，ふたつの項のMGU(Most General Unifier) を求める演算であった．これは，（未定義部分をもつ）二つの木の同型性の決定問題とみなせる．のちに，これは，二つの（未定義部分をもつ）ラベル付き有向グラフの同型性の決定問題＋オカレンスチェックとみなされるようになった．ラベル付き有向グラフの同型性の決定問題は，いわゆるUnion-Find問題の応用としてほとんど線形時間で解けることが知られている．一方，オカレンスチェックはグラフが，木であるかどうかのチェックであり，局所的な性質ではないので，並列的／分散的な処理になじまない．この拡張ユニフィケーションの並列な算法でも，停止性と一意性をもつのが普通である．

２．同期機構をもつユニフィケーション

Concurrent Prolog(CP) のような，各並列事象間の因果／呼応関係を記述するための，いわゆる，並列処理記述言語には同期機構が不可欠である．CPの場合は，Read-Only Annotationがこれにあたる．もし，これをユニフィケーションの拡張とみなした場合，どのように定義を拡張するかが問題となる．以下ではRead-Only Annotaion をふくむ，ラベル付き有向グラフの停止性，一意性をもつ並列ユニフィケーション算法をのべる．

まず、はじめに注意したいのは、このような拡張ユニフィケーションでは，ユニファイ可能／不可能（成功／不成功）の他に「情報不足のため未決」という場合がありうる事である．すなわち，算法の結果は２値から３値になる．

また，各ノードのとりうる状態も次のように三つになる．

１）未定義ノード

２）既定義ノード

３）他の未定義ノードから情報が送られて来る事を示す受信ノード

上の１）と３）の型のノードは，１対１に対応していて，その対応／逆対応をROとRWで示す事にする．すなわち未定義ノードｖについて，RO(v) は受信ノードであり，さらにRW(RO(v)) はｖ自身である．算法の記述を簡単にするために，これ以外の場合，RO(v) もRW(v) もｖ自身を表わすことにする．

この算法がループもありうる一般グラフを扱う以上，未定義変数X とそのRead-Only An

notation X? が（まわりまわって）ユニファイされる場合も配慮するべきである．例として，ふたつの項f(X?,X,X) とf(Y,Y,a)との並列ユニフィケーションを考えよう．もし，左から右の順にユニファイが行なわれた場合，X(=X?)と定数ａとのユニフィケーションが最後に行なわれる．右から左の順にユニファイされた場合，ユニフィケーションが成功する事に注意すれば，一意性を持たすために，左からの場合もユニファイできるべきである．すなわちX とX?がユニファイされた場合，すべてのX?及びX の出現は未定義ノードのごとくに振る舞うべきである．

３．算法

以下の算法はUNION-FIND問題の算法の応用である．

算法の途中の状態は，次の二つからなる．

ア）すべてのノードの集合のある類別（この各要素は（同値）類である．類はノードの集合である）

イ）ユニファイすべきノード（＝類の代表元）の対の集合LIST．

各類別について各ノードV に対してそれを含む類をCLASS(V)で表わす．

初期の類別では，各ノードV に対してCLASS(V)は，V 自身だけからなる集合である．

以下の算法UNIFY は各類がいつでも次の三つの場合のいずれかの状態のただひとつになっているように定義されている．

１）未定義類：次の既定義類でも受信類でもない類

２）既定義類：少なくとも一つの既定義ノードを含む類

３）受信類：少なくとも一つの受信ノードV についてV は含むが RW(V)は含まない類

ある類C について，類RO(C) を，次のように定義する．

もしC 中の未定義ノードV でRO(V) はC に含まれないものがあったらRO(C) は，CLASS(RO(V))を表わす．もし，そのような未定義ノードがなければ、RO(C) はC 自身とする．

算法UNIFY は，RO(C) が一意に定まるように定義されている．実はさらに，既定義類または受信類C については，RO(C)=C となる．

算法 UNIFY:

以下の操作をLISTが空集合になるか，取り除けない対だけになるか，下記の２．３．２の場合が起こるまでくりかえす．

操作ONE_STEP：

１）LISTからひとつの対＜V1，V2＞を，取り出す．

２．１）もしCLASS(V1)=CLASS(V2) ならば，この対をLISTから取り除く．

２．２）その他の場合で，もし CLASS(V1)または CLASS(V2)が未定義類ならば，この対をLISTから取り除き CLASS(V1)と CLASS(V2)を融合し， RO(CLASS(V1))と RO(CLASS(V2))を融合する．

２．３）その他の場合で，もし CLASS(V1)および CLASS(V2)が既定義ノードV およびV'をそれぞれ含むならば，

２．３．１）もし VとV'との関数記号が一致している場合，

２．３．１．１）この対をLISTから取り除き CLASS(V1)と CLASS(V2)とを融合した後，

２．３．１．２）各アーギュメントの類の対 <V11,V21>,...,<V1n,V2n>をLISTに加える．
２．３．２）その他の場合，すなわち関数記号が一致しない場合，ユニファイ不可能であり，算法を終える．
２．４）その他の場合，すなわち， CLASS(V1)も CLASS(V2)も未定義類でなく，かつ，少なくとも一方は，受信類で，かつ， CLASS(V1)と CLASS(V2)とは異なる場合，（この対は取り除けないので）何もしない．

判定：
もしLISTが空になったら，ユニフィケーションは成功である．
LISTが空ではなく，かつ，取り除けない対だけになったら，ユニフィケーションは，情報不足のため未決である．

実行例として上記の例を，取り上げる．
初期状態での類別R0とLIST0 とは、次の通り：
R0={ {f(R0(X),X,X)}, {R0(X)}, {X}, {f(Y,Y,a)}, {R0(Y)}, {Y}, {a} }
LIST0={ {f(R0(X),X,X), f(Y,Y,a)} }.
LIST0 のただ一つの元は関数記号の一致する既定義類の対なので操作ONE_STEPの結果は次の通り：
R1={ {f(R0(X),X,X), f(Y,Y,a)}, {R0(X)}, {X}, {R0(Y)}, {Y}, {a} }
LIST1={ {R0(X), Y}, {X, Y}, {X, a} }.
LIST1 の元を左から右の順に取り扱った場合の途中結果の類別R2,...,R4 は，次の通り：
R2={ {f(R0(X),X,X), f(Y,Y,a)}, {R0(X), R0(Y), Y}, {X}, {a} },
R3={ {f(R0(X),X,X), f(Y,Y,a)}, {R0(X), X, R0(Y), Y}, {a} },
R4={ {f(R0(X),X,X), f(Y,Y,a)}, {R0(X), X, R0(Y), Y, a} }.
（R3において，類{R0(X), X, R0(Y), Y}は未定義類である．）
LIST1 の元を右から左の順に取り扱った場合の途中結果の類別R2',...,R4' は，次の通り：
R2'={ {f(R0(X),X,X), f(Y,Y,a)}, {R0(X), X, a}, {R0(Y)}, {Y} },
R3'=R4'=R4.

注意０．上記の２．２の場合を融合される類で場合分けすると次のようになる：
２．２．１二つの未定義類C1,C2 の場合，C1とC2及びR0(C1)とR0(C2)が融合されそれぞれ新しい未定義類と受信類とになる．
２．２．２未定義類C1と既定義類C2の場合，C1とR0(C1)とC2(=R0(C2)) とが融合されて既定義類になる．
２．２．３未定義類C1と受信類C2の場合，C1とR0(C1)とC2(=R0(C2)) とが融合されて受信類になる．
ノードが変化するのでなく，その属する類が変化することに注意しよう．

注意１．UNIFY は必ず停止する．なぜなら，類別に対する操作は融合だけなので，その要素の数は単調に減少する．類別が変化せずに，ONE_STEPが繰り返されるときは，LISTの要素の数がへる．

注意２．UNIFY が失敗または未決になった場合，類別自身は一意にはならない．例えば，f(X?,Y?)とf(Z,Z)とのユニフィケーションは未決であり，Z がX?またはY?のどちらと同じ類になるかはLISTの選択順序に依存する．ここでの一意性は，成功／失敗／未決の３値についてである．もっとも，成功の場合には，類別も一意になる．

これらの一意性により，並列処理中の局所環境に付け加わる情報の相対的到着順序が結果に影響しない事が保証される．

注意３．このUNIFY は普通のユニフィケーションの自然な拡張になっている．実際，受信ノード／受信類を省けば，普通のユニフィケーションになる．また，QuteやGuarded Horn Clausesで採用された「外部変数への代入制限付きユニフィケーション」も，受信ノード／受信類の代わりに外部参照ノード／外部参照類を導入する事により記述できるであろう．

注意４．上に定義したRead-Only Annotationのユニフィケーションの異常なのは，受信類が更に融合されて未定義類に回復する場合がある点である．これは，上記の一意性の要請から生じたことであるが，単なる異常な場合というだけではなく，単一環境における二進セマフォーの直接的な実現ができるという実用的な（？）価値をもっている．セマフォーをプロセスではなく，未定義変数で表現するのである．P 命令は，局所的な受信ノードX?をセマフォー変数にユニファイすることで表現される．V 命令は，X?とX とのユニフィケーションによる．

もっとも，普通のCPの実現では親環境との整合性チェック（＝ユニファイ可能性）をCommitしてから行なうので，このセマフォーの直接的実現はうまくいかない．

pure Prolog with Read-Only Annotation とでもいうべきCPのサブセットを，考えればいいかもしれない．この場合，親環境と整合しないとCommitできない事にしておく．

この言語は，Shapiro のFlat Concurrent Prologに似ているが，（ヘッド部のユニフィケーション以外の）ガード部を持たないというより「過激な」サブセットになっている．

CPやGHC と比較した場合，動作の中間状態が単なる未解決ゴールの列で表現できるのが利点であろう．CPやGHC ではガード部の分解状態の表現のために全体としての中間状態は木状になってしまう．

もっとも，ガード部なしではプログラムが書きにくくなるので，なんらかのシンタックスシュガーは必要であろうが，これは，メタ記述と併せて導入したい．

Parallel Object Oriented Programming

with

Colored Messages

Masami Hagiya

Research Institute for Mathematical Sciences
Kyoto University

ABSTRACT

1. Introduction

There have been proposed many formalisms of parallel object oriented programming based on the actor theory (see [1]). In this short paper, we present a formalism in which each message has a color and the color allows an object to send a message to itself recursively. This approach completely unifies the object as a multiple-entrance procedure and the object as an independent computational process.

We have built an experimental system according to our formalism on Common Lisp (see [2]). The reason of the selection of Common Lisp is that it completely supports the functional closure.

2. Message

A message consists of the following five components:

* target object
* color
* message identifier
* arguments
* continuation.

The target object is the object that the message is directed to. The color of a message is unique to our approach: each message has a color as one of its components. We assume that there are an infinite number of colors and a new color is always available. (The internal structure of colors is not relevant.) The message identifier is the name of the message; in a window system, for example, a message to a window may have a message identifier such as :EXPOSE or :STRETCH. (Since our experimental system is implemented on Common Lisp and Common Lisp

keywords are used for message identifiers, we prefix a colon before a message identifier.) Theoretically, the message identifier may be included in the arguments of the message, but from the practical point of view, we do not consider a message identifier as an argument. The continuation of a message is a functional closure which is invoked when the target object has completed the execution of the method that corresponds to the message identifier of the message. A message may have no continuation, in which case, the continuation is treated as NIL. In contrast to the message identifier, the continuation is included in the arguments of the message and, by convention, the keyword argument :RETURN of the message always has the continuation as its value (see [2]).

When a message arrives at an object and the object is ready to process the message, the method of the object that corresponds to the message identifier of the message is executed. The method may return a value, and the value becomes the value of the message.

3. Color

An object can process more than one message at a time, but they should be of a same color. While an object is processing a message (or messages), the object is considered to have the color of the message (it is colored with the color of the message).

When an object has no color, i.e. it is processing no message, it can process a message as soon as the message arrives at it; here, processing a message means to invoke the method of the object that corresponds to the message identifier of the message.

When an object has a color, i.e. it is processing a message or messages, messages of other colors than that of the object are queued in the order of their arrival, and messages of the same color as that of the object are processed immediately. As will be explained in Section 8, the messages that an object is processing (i.e. they are of a same color) forms a stack and only the method that corresponds to the top of the stack is active.

When the control reaches the end of the method, the object loses the color of the message; if more than one messages are being processed, the object loses the color after all the messages have been processed.

4. Object

An object is a collection of methods, each corresponding to a distinct message identifier. A method takes a form of a function; it has a lambda-list and a body (sequence of forms). An object is created

by evaluating the form:

```
(! . <method-list>),
```

where a method takes the following form:

```
(<message-identifier> <lambda-list> . <body>).
```

For example,

```
(! (:VALUE (N)
     (IF (= N 0) 1 (* (<- SELF :VALUE (1- N)) N))))
```

will return an object with one method, which, given a message with the message identifier :VALUE and the argument N, returns the factorial of N. The form beginning with <- is a message sending form, which will be explained in the sequel, and SELF is a pseudo variable that holds the object as its value. The use of SELF is not theoretically essential; we can create the object by evaluating the following LET form:

```
(LET ((SELF
       (! (:VALUE (N)
            (IF (= N 0) 1 (* (<- SELF :VALUE (1- N)) N))))))
  SELF),
```

in which SELF is explicitly bound to the object itself.

5. Message Sending Form

A message is sent to an object by either of the following forms:

```
(<- <object> <message-identifier> . <arguments>)

(<& <object> <message-identifier> . <arguments>).
```

(All the arguments in both forms are evaluated; <- and <& are functions and not special forms nor macros.) Evaluating these forms creates a message as described below.

The message created in the first form has the following components:

target:	given in the form
color:	the color of the sender object
identifier:	give in the form
arguments:	given in the form
continuation:	the continuation that receives the value of the message and computes the rest of the method of the sender object.

This continuation is constructed implicitly by the form and not explicitly supplied as an argument. The value of the message becomes the value of the form. For example, let us consider the following method

```
(:VALUE (N)
  (IF (= N 0) 1 (* (<- SELF :VALUE (1- N)) N)))
```

which contains the message sending form

```
(<- SELF :VALUE (1- N)).
```

This message sending form creates a message with the following components:

target:	the value of SELF (the sender object)
color:	the color of the sender object
identifier:	:VALUE
arguments:	the value of (1- N) and the continuation
continuation:	the following continuation

```
(!! M (-> RETURN (* M N))).
```

The continuation is explained in Section 7. Since the rest of the method :VALUE is passed to the target object as a continuation, the control of the sender object suspends until the continuation is invoked by the target object (or other objects). This form corresponds to what is called 'now type' in [1].

In the second form, the message has the following components:

target:	given in the form
color:	newly created
identifier:	given in the form
arguments:	given in the form
continuation:	the value of the :RETURN keyword argument in the form, or NIL if not supplied.

The form has no value (or has value NIL) and the control of the sender object does not suspend. This form corresponds to what is called 'past type' in [1].

6. Method

A method takes the following form:

```
(<message-identifier> <lambda-list> . <body>).
```

If the lambda-list has the :RETURN keyword argument, it is given the continuation of the message. In this case, the continuation should be called by the continuation calling form (see the next section), if it is necessary. As an example, let us consider making a binary semaphore.

```
(LET ((Q (MAKE-Q)))
  (! (:P (&KEY RETURN)
       (IF (QENDP Q)
           (-> RETURN NIL)
           (ENQ RETURN Q)))
     (:V ()
       (IF (QENDP Q)
           NIL
           (-> RETURN (DEQ Q))))))
```

The method :P has the :RETURN keyword argument, while :V does not. The forms beginning with -> are continuation calling forms and will be explained in the next section. In the above example, we assume that MAKE-Q makes a queue (which may be realized by a list and a pointer that points to the last cell of the list), ENQ enqueues an element in a queue, DEQ dequeues an element from a queue and returns the dequeued element, and QENDP checks if a queue is empty.

If the lambda-list does not have the :RETURN keyword argument and the continuation of the message is not NIL, the value of (the last form of) the body of the method is (implicitly) applied to the continuation of the message after the control of the method reaches its end; see the :V method above. (If the continuation is NIL, nothing happens.) This convention may seem quite ad-hoc, but very useful in practice.

7. Continuation

For explicitly creating a continuation, the following form is prepared:

(!! <variable> . <body>),

which creates a continuation that returns the value of (the last form) of the body when called with the (initial) value of the variable. (The variable is, of course, local to the continuation.)

For (explicitly) calling a continuation, the following form is prepared:

(-> <continuation> <value>),

which calls the continuation with the value as the (initial) value of the variable of the continuation, if the continuation is not NIL. The

continuation calling form returns the value of (the last form of) the continuation. (If the continuation is NIL, the value is NIL.)

A continuation can be called only once. When a continuation is called, it commits a suicide after evaluating its body. This point is vague in most of the formalisms of parallel object oriented programming. We think that the suicide condition should be explicitly stated in the semantics of the language, because with the suicide condition, the implementation becomes quite simple for implicit continuations.

8. Discussions

Since the message sending form is restricted to the two types given in Section 5, at any instance of the execution, there is at most one active object colored with each color. It means that a color may be considered as a process (having its own processor and stack) that travels the world with objects as stepping stones. The second message sending form creates a new color; it creates a new process and initiates it with the message.

Each object may send a message to itself, directly or indirectly, as far as the color of the message is the same as that of the object; i.e. recursion is possible in our formalism. It means that an object can play the role of a recursive procedure in the ordinary sequential language. In this sense, our formalism generalizes the ordinary (recursive) procedural language to the parallel one.

It is obvious that simple loops are realized by sending a message recursively. For example, the iterative version of factorial may be as follows:

```
(! (:VALUE (N) (<- SELF :LOOP N 1))
   (:LOOP (I F)
     (IF (> I 0)
         (<- LOOP (1- N) (* F N))
         F))).
```

The recursive call can be easily converted to a loop by the technique of tail-recursion optimization. From the practical point of view, however, the construct of loop should be included as a primitive one. Here, we just wanted to show the generality of our approach.

We can extend the formalism so that the colors may have priorities. Special colors (e.g. emergency colors) may be defined and given the highest priority; it will be processed even if the object is colored with another color.

In a multi-processor environment, there are at least two methods to

implement our formalism. The first one is to assign a processor to each color as discussed above. This is more suited to a one-processor multi-programming environment. But, in a multi-processor environment, each process should access the shared memory in which the code of each object resides.

In the second method, a processor is assigned to each object. An object has a local memory and a stack; the code of the object is placed in the local memory, and a new frame is constructed on the stack, each time the object is invoked.

9. Hamming's Problem

As an example, we give the solution to the Hamming's famous problem: to list all the numbers of the form 2**i*3**j*5**k in the increasing order. First, we define a function MULTIPLIER-CHANNEL that returns an object with two methods :PUT and :GET.

```
(DEFUN MULTIPLIER-CHANNEL (M)
  (LET ((Q (MAKE-Q))
        (R (MAKE-Q)))
    (! (:PUT (N)
         (IF (QENDP R)
             (ENQ N Q)
             (-> (DEQ R) N)))
       (:GET (&KEY RETURN)
         (IF (QENDP Q)
             (ENQ RETURN R)
             (-> RETURN (DEQ Q)))))))
```

The merger object can be created by evaluating the following form:

```
(LET ((M2 (MULTIPLIER-CHANNEL 2))
      (M3 (MULTIPLIER-CHANNEL 3))
      (M5 (MULTIPLIER-CHANNEL 5))
      (X2 2)
      (X3 3)
      (X5 5)
      (N 100)
      MIN)
  (! (:LOOP ()
       (SETQ MIN (MIN X2 X3 X5))
       (<& M2 :PUT MIN)
       (<& M3 :PUT MIN)
       (<& M5 :PUT MIN)
       (IF (= MIN X2) (SETQ X2 (<- M2 :GET)))
       (IF (= MIN X3) (SETQ X3 (<- M3 :GET)))
       (IF (= MIN X5) (SETQ X5 (<- M5 :GET)))
```

```
(IF () (DECF N) 0) ((<- SELF :LOOP))))).
```

This will return an object with one method :LOOP. After receiving the message :LOOP, it will successively supply first 100 numbers to the three multiplier-channels.

Note that we are assuming that the order of the messages sent from the same sender are kept, if they arrive at the same target.

10. Simulation on Common Lisp

In the simulation on Common Lisp, an object is represented by a 3-tuple with

- message queue
- message counter
- closure.

The message queue queues the messages that have arrived at the object. The message counter counts how many messages (of the same color) are being processed in the object. The closure is the code of the object represented by a functional closure of Common Lisp.

A continuation is represented by a 2-tuple with

- alive flag
- closure.

The alive flag keeps whether the continuation is alive or has committed a suicide.

! is implemented as a macro, which is expanded into a complicated closure-generating form. For example, the object

```
(! (:VALUE (N)
     (IF (= N 0) 1 (* (<- SELF :VALUE (1- N)) N))))
```

expands to something like

```
(LAMBDA ()
  (LET ((M77 (OBJECT-NEXT-MESSAGE SELF)))
    (CASE (MESSAGE-ID M77)
      (:VALUE
        (APPLY
          #'(LAMBDA (N &KEY RETURN)
              (IF (= N 0)
                  (-> RETURN 1)
                  (QSEND SELF
```

```
                                (CAR (OBJECT-STACK SELF)) (1- N)
                                :RETURN
                                (!! R83 (-> RETURN (* R83 N))))))
                    (MESSAGE-ARGS M77)))
                (T (ERROR "THE MESSAGE ~S IS UNDEFINED."
                          (MESSAGE-ID M77))))))).
```

Note that the implicit :RETURN argument and the implicit continuation (form) are generated. OBJECT-NEXT-MESSAGE returns the next message in the message queue.

!! is also a macro, but simpler than !.

Objects and continuations are both represented by a closure with no argument (see above).

<- and <& queues the object and the message to a queue called the closure queue. -> queues the continuation and the value.

The dispatcher repeatedly dequeues a closure (representing an object or a continuation) from the queue and calls it.

References

[1] Akinori Yonezawa, Hiroyuki Matsuda, Towards Object Oriented Concurrent Programming, Kokyuroku 547, Research Institute for Mathematical Sciences, Kyoto University.

[2] Guy L. Steele Jr., Common Lisp, the Language, Digital Press, 1984.