並行論理プログラムにおける逐次実行部分の抽出方法論

加藤紀夫[†]　上田和紀[††]

並行言語のコンパイル技法として従来，ソースコードおよび中間言語コードに対する等価変形が主に研究されてきた．しかし，実際に最適化コンパイラを作成するにはそれだけでは不十分で，いかに特定の最適化技法を適用するかということが重要になってくる．例えば，ヒープの破壊的代入やループ最適化実行などの効果的な最適化技法は，プログラム変形では保証できない．今までこれらはメモリ管理を考えないソースコード上の解析によって行われてきたため，とくに大域的な最適化に対して最適化技法を理論づける方法が分からないことが多かった．また，変数のタグ除去やループ融合など，解析のモジュール性が要求される最適化技法には，決定版となる解析法が存在しなかった．

本論文では，データフローに注目したボトムアップな解析によって，以上の問題点を解決しながら最適化された目的コードの生成を行う方法について述べる．このボトムアップな解析は，プロセスの動作に関する期待を表すインタフェースを利用して行われる．インタフェースを表示的意味論を使って定式化する方法とともに，それによって最適化技法の安全性を保証することができる理由も示す．

A Methology for Extracting Sequential Execution Fragments in Concurrent Logic Programming Languages

Norio Kato[†] and Kazunori Ueda[††]

The researches on compiling techniques for concurrent languages so far mainly have focused on equivalent transformations of source codes and intermediate language codes. However, to build a practical optimizing compiler requires another thing, namely, a solution for how to apply particular optimizing techniques. In fact, some effective optimizing techniques such as heap destructive assignment and loop optimized execution cannot be guaranteed by program transformation. The analyses for these techniques tend to have been on source codes without taking account of memory management, which often obscures how to justify themselves particularly when they are applied to global optimization. Moreover, no definitive methods have been existed to analyze those optimizing techniques which require the modularity of analyses such as variable unboxing and loop fusion.

This paper presents a method to generate optimized object codes directed by a bottom-up analysis that focuses on dataflow, with above problems solved. The analysis is performed with *interfaces*, where an interface represents an expectation of the execution of a process. We will show how to formalize interfaces in terms of denotational semantics as well as the reason why such formalization can justify an optimizing technique.

1. はじめに

1.1 背　景

並行論理型言語は，その非決定性と並列性から，その処理系は多数の細粒度プロセスが同期を取りながら並列実行できればそれでよいと思われている．しかし実際には並行論理プログラムにおいても，例えば以下のように，逐次的に実行する方がよい部分が存在する．

(1) そもそもアルゴリズムに並列性がない場合．

(2) 並列実行できる部分（future）の粒度があまり大きくないため，並列実行すると通信遅延や同期処理で確実に遅くなる場合．

(3) 並行プロセス間のメッセージフローを，よりメモリアクセスが少ないコントロールフローとして表現できる場合．

このようなプロセスの実装を一般的な実装方法で行うと，並行処理に伴う不必要なオーバーヘッドが生じる．

例えば図1のような並行論理プログラムで，intlist と sum を交互に実行できることが解析できれば，以下のような最適化コンパイルが可能になると思われる．

- タグ除去（unbox 化）
- クロージャ（引数リスト）の共有
- 局所変数のレジスタによる実装
- メッセージフローのコントロールフローによる実装（メッセージ指向コンパイル）

そのような解析のうち最も重要となるのが，プロセス

† 早稲田大学大学院理工学研究科
Graduate School of Science and Engineering, Waseda University
†† 早稲田大学理工学部
School of Science and Engineering, Waseda University

を中断せずに実行できることを解析することである．

従来これらの最適化を行うために，ゴール間の依存解析，ゴールの抽象実行，述語引数のモード解析，および出力要求から必要な入力量を求める解析などが考えられてきた．しかしこれらは，メモリ管理を考えないソースコード上の解析であることや，最適化技法自身のモジュール性の低さのために，とくに大域的な最適化に対して最適化技法の安全性をうまく説明できないことが多かった．そのため，試作される最適化コンパイラは信頼性を欠いたり，逆に安全のために自明な場合しか扱えないことが多かった．このような事情から，これらの最適化のための一般性のある決定版となる解析方法や，これらの解析に対して最適化技法の安全性を一般的に理論づける定式化の方法論は，今まで存在しなかった．

1.2 本論文の特徴

プログラムに対する最適化技法の適用は，実際に技法を適用する作業と，その安全性を保証する作業からなる．本論文は，並行言語の最適化コンパイラの構成法として，意味論に支えられたインタフェースという定式化を用いて，これらの作業を行う方法を提案する．

並行実行される単位をプロセスと呼ぶことにする．プロセスに対して我々は，特定の入力を仮定したときに，特定の出力をした後に特定のプロセスとして動作するという性質を論じることができる．この性質を，プロセスのインタフェースと呼ぶことにする．インタフェースは，プログラマがそのプロセスの動作に対して抱く期待を表していると考えられる．そのような期待はプログラムを宣言的に読むことによってボトムアップに解析することができるため，インタフェースの解析は本質的に高いモジュール性を持っている．しかもこの解析は同時に，最適化コンパイルのために必要な情報を解析する作業にも対応しており，プログラム中に存在する逐次実行部分を抽出し，その内部のデータフローに従って最適な目的コードを生成することができる．

我々は，インタフェースを定式化するために表示的意味論を利用する方法を提案する．これによって，プロセスがそのような性質を満たすことを定式化することができるとともに，特定の条件を満たすプロセスに対してその性質を体系的に証明することもできるようになる．

そのために，中断と終了を扱う非決定的プロセスの簡素な表示的意味論を導入し，その意味論が具体的な最適化技法の安全性を保証する理由について説明する．意味論によって最適化技法の安全性が体系的に証明できることによって，最適化コンパイラは信頼性と拡張性を獲得する．それを実証するために，ループ最適化実行などを実際に適用するコンパイル手順の具体例を示す．

1.3 本論文の構成

始めに対象とする言語の形式的な定義を与え（2 節），その表示的意味論を定義する（3, 4 節）．次にその表示的意味論を使ってインタフェースを定式化し，それを使ったボトムアップな解析の手順を紹介する（5 節）．この解析を使用して実際に効率のよい最適化コンパイラを構成するために，述語のクラス分けを定義すると同時に，クラス分けの求め方も示す（6 節）．最後に，例を使って逐次実行部分の最適化コンパイルの流れを示し，本方法論の有効性を検証する（7 節）．

なお 3 節および 4 節は，1 つの表示的意味論を与えるだけであるため，応用的には読み飛ばして構わない．

2. プログラム言語

本節では，研究対象とする並行論理型言語を定義する．並行論理型言語は，並行に動作する各プロセスが，単調に情報が増える共有ストアに対して制約の入出力を繰り返すことによって計算を表現する．制約の出力はストアを増強し，制約の入力はストアの強さの検査に対応する．

2.1 ストアの空間

$\langle D, \leq \rangle$ を $true$ を最小元，$false$ を最大元とする完備束とする．D の要素を**制約**と呼ぶ．$c, d \in D$ のとき，上限 $c \sqcup d$ は，制約 c および d の論理積を表すものとする．換言すれば $c \leq d$ は，d が c より多くの情報を持っていることを表すものとする．また $true$ は情報が無いことを表すものとし，$false$ は情報の矛盾を表すものとする．

D は，制約の集合であるとともに，プログラムの実行に従って情報が単調に増えるストアの集合でもある．具体的には，制約情報 c を持つストアは，制約 d の追加（tell）によってそれらの論理積 $c \sqcup d$ を持つようになる．

$\uparrow c \stackrel{\text{def}}{=} \{d \in D \mid d \geq c\}$ とする．

変数記号の可算集合 Var と，そのスーパーセットである**項**の集合が与えられているとする．このとき D は，下記の公理のように，2 つの項が単一化されていること

```
b  ::  stairs(N,X)      :- true    | intlist(1,N,S), sum(S,0,X).
c1 ::  intlist(K0,N,S)  :- K0 >=N  | S=[].
c2 ::  intlist(K0,N,S)  :- K0 < N  | S=[K0|S1], K:=K0+1, intlist(K,N,S1).
d1 ::  sum([],   X0,X)  :- true    | X:=X0.
d2 ::  sum([E|S],X0,X)  :- true    | X1:=X0+E, sum(S,X1,X).
```

図 1: 並行論理プログラムの例

を表す制約を表現できるものとし，変数記号 X に対して X を隠蔽する演算子 $\exists_X : D \to D$ の存在を仮定する．

公理 2.1 2 つの項 s, t の単一化を表す制約 $(s = t)$ は D の要素であり，項 s, t および変数記号 X に対して以下の性質を満たすものとする．

(E1) $(s = s) = \mathit{true}$

(E2) $(s = t) = (t = s)$

(E3) $(X = t) \sqcup P[X] = (X = t) \sqcup P[t]$

(E4) $(f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)) \geq (s_i = t_i)$

(E5) $(f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n)) = \mathit{false}$,
if $(f, m) \neq (g, n)$

ここで $P[\cdot]$ は，項を引数に取り，D の要素であるような一階原子論理式を決定するコンテキストとする． □

公理 2.2 $\exists_X$ 演算子は，以下の性質を満たすとする．

(X1) $\exists_X(c) \leq c$

(X2) $c \leq d \Rightarrow \exists_X(c) \leq \exists_X(d)$

(X3) $\exists_X \exists_Y(c) = \exists_Y \exists_X(c)$

(X4) $\exists_X(c \sqcup \exists_X(d)) = \exists_X(c) \sqcup \exists_X(d)$ □

公理 2.2 は，一般の制約に対して変数記号の隠蔽が満たすべき性質を定式化している．

さらに，単一化と隠蔽の関係を規定する下の公理 2.3 によって，$\exists_X$ が存在限量子の効果を表すようになる．

定義 2.1

(V1) $\mathit{var}(c) \stackrel{\text{def}}{=} \{X \in \mathit{Var} \mid \exists_X(c) \neq c\}$

(V2) $\exists_{\{X_1, \ldots, X_n\}}(c) \stackrel{\text{def}}{=} \exists_{X_1} \ldots \exists_{X_n}(c)$

(V3) $\overline{\exists}_X(c) \stackrel{\text{def}}{=} \exists_{\mathit{var}(c) \setminus \{X\}}(c)$

$\mathit{var}(c)$ は, 直観的には，制約 c に自由に出現する変数記号の全体集合を表す．また $\overline{\exists}_X(c)$ は，制約 c の中で変数記号 X に関係する部分のみに対応する制約を表す．

公理 2.3 項 s, t および変数記号 X, Y に対して，以下が成り立つとする．

(EX1) $X \notin \mathrm{var}(t) \Rightarrow \exists_X(X = t) = \mathit{true}$

(EX2) $X \notin \mathrm{var}(s) \cup \mathrm{var}(t) \Rightarrow \exists_X(s = t) = (s = t)$

(EX3) $X \not\equiv Y \Rightarrow c \leq (X = Y) \sqcup \exists_X(c \sqcup (X = Y))$

ただし項 t に対して $\mathrm{var}(t)$ は，項 t に出現する変数記号の全体集合を表すものとする．(EX3) は，変数記号の読み替えに関する D の構造の不変性を表し，2.2 節において変数変換の定義を可能にする． □

例

$(X = 1) \leq (X = 1) \sqcup (Y = 1) = (X = Y) \sqcup (Y = 1)$

$\exists_X \mathit{false} = \exists_X(1 = 2) = (1 = 2) = \mathit{false}$

$\mathit{var}(X = \mathrm{f}(Y)) = \{X, Y\}$ □

2.2 変数変換

述語呼び出しにおける変数名の付け替えを表現するために，変数変換という記法を導入する．本論文では，子のストアを親のストアに変換するために変数変換を使用する☆．変数変換は典型的に σ と表記される．

s および t を項とする．$\sigma = \{s \mapsto t\}$ とおくとき，**変数変換** σ を以下のように定義する．

$$\sigma(c) \stackrel{\text{def}}{=} \exists_X(\overline{\exists}_X(c \sqcup (X = s)) \sqcup (X = t))$$

ここで X は他に現れない変数記号とする．詳細に言えば，$X \notin \mathit{var}(c) \cup \mathrm{var}(s) \cup \mathrm{var}(t)$ とする．また σ に対して $\sigma^{-1} \stackrel{\text{def}}{=} \{t \mapsto s\}$ と定義する．

定理 2.1 $c \leq d \Rightarrow \sigma(c) \leq \sigma(d)$ □

定理 2.2 $\sigma(\sigma^{-1}(c) \sqcup d) \sqcup c = \sigma(d) \sqcup c$ □

本論文では今後とくに $\alpha \in \mathit{Var} \setminus \mathrm{var}(t)$ を満たす α, t に対して $\{\alpha \mapsto t\}$ の形をしている変数変換を考え，これを典型的に σ と表記する．変数変換 $\{\alpha \mapsto t\}$ は，仮引数 α を持つ述語 p があったとき，ゴール $p(t)$ のリダクションによる子が親に対して行う出力を表すために使用される．一方 σ^{-1} は，親が子に対して行う入力を表すために使用される．この σ に対して以下が成り立つ．

- $\sigma(c) = \exists_\alpha(\overline{\exists}_\alpha(c) \sqcup (\alpha = t))$
- $\sigma^{-1}(c) = \overline{\exists}_\alpha(\exists_\alpha(c) \sqcup (\alpha = t))$
- $\sigma(c \sqcup \overline{\exists}_\alpha d) = \sigma(c) \sqcup \sigma(d)$
- $\sigma^{-1}(\sigma(c)) = \sigma^{-1}(\mathit{true}) \sqcup \overline{\exists}_\alpha(c)$

以上の公式は，本論文の証明において使用される．

例

$\sigma = \{\alpha \mapsto \langle 1, Y \rangle\}$ のとき，

$\sigma(\exists_B(\alpha = \langle 1, \mathrm{f}(B) \rangle)) = \exists_Z(Y = \mathrm{f}(Z))$. □

2.3 エージェント

述語記号の集合 Pred に対して，プロセスに対する構文クラスである**エージェント**を次のように定義する．エージェントの全体集合を Agents と表記する．

$$A ::= \mathbf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i \mid A \| A \mid \exists X(A) \mid p(t)$$

ただし $c \in D$, $n \geq 1$, $p \in \mathit{Pred}$ とし，t は変数記号 α が現れない項とする．X は α 以外の変数記号とする．

また，エージェント A に対して，A に自由に出現する変数の全体集合 $\mathrm{fv}(A)$ を以下のように定義する．

$$\begin{aligned}
\mathrm{fv}(\mathbf{tell}(c)) &\stackrel{\text{def}}{=} \mathit{var}(c) \\
\mathrm{fv}(\textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i) &\stackrel{\text{def}}{=} \textstyle\bigcup_{i=1}^{n} \mathit{var}(c_i) \cup \mathrm{fv}(A_i) \\
\mathrm{fv}(A_1 \| A_2) &\stackrel{\text{def}}{=} \mathrm{fv}(A_1) \cup \mathrm{fv}(A_2) \\
\mathrm{fv}(\exists X(A)) &\stackrel{\text{def}}{=} \mathrm{fv}(A) \setminus \{X\} \\
\mathrm{fv}(p(t)) &\stackrel{\text{def}}{=} \mathrm{var}(t)
\end{aligned}$$

2.4 プログラムの定義

プログラム Prog は，述語記号からエージェントへの写像 $\mathit{Prog} : \mathit{Pred} \to \mathit{ReducingAgents}$ とする．ただしそのエージェントでは変数記号 α が仮引数と解釈され，それ以外の変数記号は自由に出現してはならないとする．すなわち，次のように定義する．

$$\mathit{ReducingAgents} \stackrel{\text{def}}{=} \{A \in \mathit{Agents} \mid \mathrm{fv}(A) \subset \{\alpha\}\}$$

未定義の述語 p に対しては，$\mathit{Prog}(p) = \mathbf{ask}(\mathit{false}) \to \mathbf{tell}(\mathit{true})$ などと定義するとよい．

例

非決定的にリストを併合する述語 merge は，次のよ

☆ 変数変換は論文 7) で最初に導入されたが，そこでは，親のストアを子のストアに変換するために使用された．

うに表すことができる．

$$\begin{aligned}
&\mathit{Prog}(\mathrm{merge}) = \exists\alpha_1\exists\alpha_2\exists\alpha_3(\alpha = \langle\alpha_1,\alpha_2,\alpha_3\rangle \parallel (\\
&\quad \mathbf{ask}(\alpha_1 = []) \to \mathbf{tell}(\alpha_3 = \alpha_2)\\
&\quad + \mathbf{ask}(\alpha_2 = []) \to \mathbf{tell}(\alpha_3 = \alpha_1)\\
&\quad + \mathbf{ask}(\alpha_1 = [_|_]) \to \exists A\exists X\exists Z(\,\mathbf{tell}(\alpha_1 = [A|X])\\
&\qquad \parallel \mathbf{tell}(\alpha_3 = [A|Z]) \parallel \mathrm{merge}(X,\alpha_2,Z)\,)\\
&\quad + \mathbf{ask}(\alpha_2 = [_|_]) \to \exists A\exists Y\exists Z(\,\mathbf{tell}(\alpha_2 = [A|Y])\\
&\qquad \parallel \mathbf{tell}(\alpha_3 = [A|Z]) \parallel \mathrm{merge}(\alpha_1,Y,Z)\,)))
\end{aligned}$$

ただし $(X = [\_|\_]) = \exists_Y\exists_Z(X = [Y|Z])$ とする． □

2.5 遷移系の定義

遷移系では，表示的意味論との関係を密接にするために，エージェントの構文を拡張する必要がある．とくに $\exists X(A)$ エージェントは，X に関する情報を保持するための内部ストアを持つと考えるのが一般的となっている．そこで，遷移系を定義するためにエージェントの構文を拡張した構文クラス *ConfAgents* を次のように定義する．

$$A' ::= A \mid \mathbf{stop} \mid A' \| A' \mid \exists X(A',c) \mid \sigma(A',c)$$

ただし A はエージェントとし，$c \in D$ とする．X は α 以外の変数記号とする．σ は変数変換とする．

処理系のコンフィギュレーションの全体集合 *Conf* を，$\mathit{Conf} = \mathit{ConfAgents} \times D$ と定義する．

プログラム *Prog* に対して，操作的意味論を次の公理を満たす $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ の最小集合と定義する．

(R1) $\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{stop}, c \sqcup d\rangle$

(R2) $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, d\rangle \longrightarrow \langle A_i, d\rangle,\ \text{if } c_i \le d$

(R3) $\langle p(t), d\rangle \longrightarrow \langle \{\alpha \mapsto t\}\,(\mathit{Prog}(p), \mathit{true}), d\rangle$

(R4) $$\frac{\langle A, c \sqcup \sigma^{-1}(d)\rangle \longrightarrow \langle A', c'\rangle}{\langle \sigma(A,c), d\rangle \longrightarrow \langle \sigma(A',c'), d \sqcup \sigma(c')\rangle}$$

(R5) $$\frac{\langle A, \exists_X d\rangle \longrightarrow \langle A', c'\rangle}{\langle \exists X(A), d\rangle \longrightarrow \langle \exists X(A',c'), d \sqcup \exists_X c'\rangle}$$

(R6) $$\frac{\langle A, c \sqcup \exists_X d\rangle \longrightarrow \langle A', c'\rangle}{\langle \exists X(A,c), d\rangle \longrightarrow \langle \exists X(A',c'), d \sqcup \exists_X c'\rangle}$$

(R7) $$\frac{\langle A, d\rangle \longrightarrow \langle A', d'\rangle}{\langle A \parallel B, d\rangle \longrightarrow \langle A' \parallel B, d'\rangle}$$

$$\frac{\langle B, d\rangle \longrightarrow \langle B', d'\rangle}{\langle A \parallel B, d\rangle \longrightarrow \langle A \parallel B', d'\rangle}$$

(R3) は，新しいストアを 1 つ用意し，そのストアを使用してゴール $p(t)$ の実行を開始する計算を表す．(R5) は，同様にストアを用意して，新しい変数記号を供給する．(R4) および (R6) は，サブエージェントの計算が起こるたびに，エージェントの内部ストアと外側の共有ストアを最新の状態に更新する．(R5) は，(R6) の特別な場合と考えることができる．実際，$\exists X(A)$ は $\exists X(A, \mathit{true})$ の略記であると考えることもできる．

コンフィギュレーション $\langle A, c\rangle$ から始まる計算が存在しない場合を考える．そのような $\langle A, c\rangle$ のうち，$A \in \mathbf{Stop}$ のものは**終了**していると定義し，そうでないものは**中断**していると定義する．ただし終了していると考えるエージェントの全体集合 **Stop** は，次のように定義する．

$$S ::= \mathbf{stop} \mid S \parallel S \mid \exists X(S) \mid \exists X(S,c) \mid \sigma(S,c)$$

ただし $X \in \mathit{Var}\backslash\{\alpha\}$, $c \in D$ とし，σ は変数変換とする．

2.6 観 測 物

プログラム *Prog* とエージェント A が与えられたとき，$\langle A, \mathit{true}\rangle$ から計算を開始したときに，ストアが特定の制約を満たすようになるか，そのとき中断，終了といった計算が存在するかどうかを観測できる．我々は *Prog* に対して，A の**観測物** $\mathcal{O}[\![A]\!]$ を次のように定義する．

$$\begin{aligned}
\mathcal{O}[\![A]\!] \stackrel{\text{def}}{=}\ & \{c' \cdot \bot \mid \langle A, \mathit{true}\rangle \longrightarrow^* \langle A', c\rangle,\ c' \le c\}\\
& \cup \{c \cdot dd \mid \langle A, \mathit{true}\rangle \longrightarrow^* \langle A', c\rangle \not\longrightarrow, A' \notin \mathbf{Stop}\}\\
& \cup \{c \cdot tt \mid \langle A, \mathit{true}\rangle \longrightarrow^* \langle A', c\rangle \not\longrightarrow, A' \in \mathbf{Stop}\}
\end{aligned}$$

我々の考えている言語は，ストアへの制約追加時にストアが矛盾するかどうかを追加前に検査できない．すなわち制約追加の失敗は，ストアが *false* に過制約されることによって表現され，観測される．

上記の他に注目すべき観測として，無限の計算が存在する（**発散**している）かどうかが挙げられる．しかし上の定義で分かるように，本論文では発散を観測物として扱わない．それにもかかわらず本論文の表示的意味論は，コンパイラ最適化に役立つことが 5 節で示される．それは端的に言えば，適切な局所的選択により終了できるプロセスは，終了するようにコンパイルできるからである．

3. プ ロ セ ス

プログラムが与えられたとき，エージェントは，ストアに対する制約の入出力や観測物の生成を行うオブジェクトと考えられる．このように考えたときのオブジェクトを，プロセスと呼びたい．本節は，プロセスの表示的意味論を定義するための土台を与える．

3.1 インタラクション

2.6 節の観測物だけでは，そのプロセスの正確な意味は分からない．例えば，入力を待ってから何かを出力して終了するといった一連の動作の存在が表現できていない．ある時点までになされたストアに対する入出力の有限列を，**インタラクション**と呼ぶ．本節では，インタラクションの後に中断および終了することがあるかどうかを組み合わせて役に立つ意味論を作ることを目指す．

3.2 トレース演算子

関数 $r : D \to D$ は，以下の性質を満たすとき（Scott の）閉包演算子と呼ばれる．

(CO1) $c \le r(c)$

(CO2) $c \le d \Rightarrow r(c) \le r(d)$

(CO3) $r(r(c)) = r(c)$

閉包演算子 r は，その値域 $r(D)$ によって完全に表現できることが知られている．すなわち $r(c) = \sqcap(r(D) \cap \uparrow c)$ が成り立つ．そこで今後，値域を閉包演算子と見なす．

閉包演算子 r は，$r^{-1} = (D \setminus r) \cup \{\mathit{false}\}$ もまた閉包演算子のとき**トレース演算子**と呼ばれる．閉包演算子 r は，ストア c を $r(c)$ に増強する振舞いを表現すると考えられ，そのとき r と r^{-1} は，互いに入出力を補完し合う

2 つのインタラクションを表すことが知られている [3]．

3.3 トレース

インタラクションは入出力の列であるが，途中までで打ち切ったインタラクションを意味的に含んでいる．そこで我々は，インタラクションの作る（可算な）系列をまとめて扱うことによって，簡素な意味論を作ることを考えた．ここではその目的のため，トレースを定義する．

以下の条件を満たす 3 項組 $\langle r,s,t\rangle \in 2^D \times 2^D \times 2^D$ を**トレース**と定義する．

(T1) $r \supseteq s$
(T2) $r \supseteq t$
(T3) $s \cap t = \emptyset$
(T4) r はトレース演算子（の値域）
(T5) $t = \uparrow(\sqcap t)$

トレースの第 1 項は典型的に r と表記され，休止点 (resting points) の集合を表す．トレースの第 2 項は典型的に s と表記され，中断点 (suspension points) の集合を表す．トレースの第 3 項は典型的に t と表記され，終了点 (termination points) の集合を表す．

休止点によって表されるトレース演算子は，インタラクション系列を表す．中断点は，トレース演算子でそのストアまで到達したときに，中断する可能性があることを表す．終了点は，同様に終了の可能性を表す．

以下で，$S \subseteq D$ に対して $\overline{S} \stackrel{\text{def}}{=} D \setminus S$ とする．

例

トレース $\langle \overline{\uparrow c} \cup \uparrow d, \overline{\uparrow c}, \uparrow c \cap \uparrow d\rangle$ は，以下の計算が存在すること表す．

- ストアに関して $\overline{\uparrow c} \cup \uparrow d$ すなわち $\lambda e.\ \text{if } c \le e \text{ then } d \sqcup e \text{ else } e$ という動作をする．つまり，ストアが制約 c を満たすとき，ストアに制約 d を追加する．
- $e \in \overline{\uparrow c}$ すなわち $c \not\le e$ なるストア e で中断する．
- ストア $e \in \uparrow c \cap \uparrow d = \uparrow(c \sqcup d)$ で終了する． □

3.4 プロセス空間

トレースの集合 P のうち，以下の閉包条件を満たすものを**プロセス**と定義し，その全体集合を $Proc$ とする．

(C1) $\langle D, \emptyset, \{false\}\rangle \in P$
(C2) $\langle r,s,t\rangle \in P \text{ and } r' \supseteq r \ \Rightarrow\ \langle r',s,t\rangle \in P$
(C3) $\langle r,s,t\rangle \in P \text{ and } s \supseteq s' \ \Rightarrow\ \langle r,s',t\rangle \in P$
(C4) $\langle r,s,t\rangle \in P \text{ and } t \supseteq t' \ \Rightarrow\ \langle r,s,t'\rangle \in P$

(C2) は，いわゆる ask-more な振舞いの可能性を表す [3]．(C3) と (C4) は，可能性の表示が目的であることを反映する．(C1) は，何も観測が無い場合を表す．

あるトレース $\langle r_0,s_0,t_0\rangle$ を含む最小のプロセスは，次のように簡単に求められる．

$$Sat\langle r_0,s_0,t_0\rangle \stackrel{\text{def}}{=} \{\langle r',s',t'\rangle \mid r' \supseteq r_0,\ s_0 \supseteq s',\ t_0 \supseteq t'\}$$

ストアへの tell も中断も終了もしないプロセスを考えることができる．$\mathbf{inert} \stackrel{\text{def}}{=} \{\langle D, \emptyset, \{false\}\rangle\}$ がそのようなプロセスであり，**inert** は完備半順序集合 $\langle Proc, \subseteq\rangle$ の最小元となる．なお **inert** は必然的な発散を表す．

3.5 プロセス式

プロセス空間上には，エージェントのコンストラクトに対応した演算子が定義できる．そのような演算子を用いて特定のプロセスを表現する式を，プロセス式と呼ぶ．

(P1) $c\star \stackrel{\text{def}}{=} Sat\langle \uparrow c, \emptyset, \uparrow c\rangle = \{\langle r,\emptyset,t\rangle \mid r \supseteq \uparrow c \supseteq t\}$

(P2) $\Box_{i=1}^{n} c_i \to P_i \stackrel{\text{def}}{=} \{\ \langle \overline{\uparrow c_j} \cup r,\ g \cup (\uparrow c_j \cap s),\ \uparrow c_j \cap t\rangle \mid \langle r,s,t\rangle \in P_j,\ g \subseteq \overline{\uparrow c_j} \cap \Gamma(c_i \to P_i)_i\ \}$

(P3) $P \parallel P' \stackrel{\text{def}}{=} \{\ \langle r \cap r',\ (s \cup t) \cap (s' \cup t') \cap \overline{(t \cap t')},\ t \cap t'\rangle \mid \langle r,s,t\rangle \in P,\ \langle r',s',t'\rangle \in P'\ \}$

(P4) $\exists_X(P) \stackrel{\text{def}}{=} \bigcup_{\langle r,s,t\rangle \in P} \exists_X\langle r,s,t\rangle$

(P5) $\sigma(P) \stackrel{\text{def}}{=} \bigcup_{\langle r,s,t\rangle \in P} \sigma\langle r,s,t\rangle$

ただし

(P2′) $\Gamma(c_i \to P_i)_i \stackrel{\text{def}}{=} \overline{\bigcup_i \uparrow c_i} \cup \bigcup_i (\uparrow c_i \cap S(P_i))$, where $S(P_i) = \bigcup_{\langle r,s,t\rangle \in P_i} s$

と定義し，これを $\Box_i\, c_i \to P_i$ の**ガード**と定義する．

また，トレースに対する隠蔽 $\exists_X\langle r,s,t\rangle$ および変数変換 $\sigma\langle r,s,t\rangle$ は，それぞれ次のように定義される．

(P4′) $\exists_X\langle r,s,t\rangle \stackrel{\text{def}}{=} Sat\langle r_0, \{c \in r_0 \mid r(\exists_X c) \in s\}, \{c \in r_0 \mid r(\exists_X c) \in t\}\rangle$ where $r_0 = \{c \mid \exists_X(r(\exists_X c)) \le c\}$

(P5′) $\sigma\langle r,s,t\rangle \stackrel{\text{def}}{=} Sat\langle r_0, \{c \in r_0 \mid r(\sigma^{-1}(c)) \in s\}, \{c \in r_0 \mid r(\sigma^{-1}(c)) \in t\}\rangle$ where $r_0 = \{c \mid \sigma(r(\sigma^{-1}(c))) \le c\}$

このとき (P4′) において $r_0 = \{c \mid \exists_X c = \exists_X d,\ d \in r\}$ というよく知られた式が成り立つことに注意する．

なお $c\star$ は，混乱がない限り c と略記する．

トレース $\langle r,s,t\rangle \in P$ に対して $\sigma\langle r,s,t\rangle$ は，次のような動作の可能性を表す．外部ストアが c のとき，内部には $\sigma^{-1}(c)$ という入力が与えられる．このとき内部で r で表される振舞いを内部ストアに対して行ったとすると，その結果は $\sigma(r(\sigma^{-1}(c)))$ として外部に与えられる．この出力は，$\sigma(r(\sigma^{-1}(c))) \not\le c$ のときそのときに限り，ストアを変化させる．さらに，外部ストアが c のとき，内部に $\sigma^{-1}(c)$ という入力が与えられて中断や終了をする可能性があるときそのときに限り，外部から見て中断や終了をする可能性がある．$\sigma(P)$ は，各トレースに対するこれらの動作の可能性の和集合として表現される．

なお，D の構造によっては，上の (P4′) や (P5′) において Sat の引数に指定されている 3 項組が，必ずしもトレースにならない場合がある．より詳細に言えば，トレースであるための条件 (T4) または (T5) を満たさないことがある．その場合には Sat を，もとの定義式のまま $2^D \times 2^D \times 2^D \to Proc$ に拡張する必要がある．

3.6 プロセスの例

プロセスを構成するトレースのうち，ある特定の手順で分岐を選択した場合を表現するトレースの全体は，そのうち最も情報量の多い 1 つのトレースに Sat を適用することにより求めることができる場合がある．

例えば，$true \to c \,\Box\, c \to d \to d$ というプロセスには，

2 つの選択の場合が存在し，$true \to c \,[\!]\, c \to d \to d = Sat\langle \uparrow c, \emptyset, \uparrow c\rangle \cup Sat\langle D, \uparrow c \cap \overline{\uparrow d}, \uparrow c \cap \uparrow d\rangle$ が成り立つ．

前者のトレース $\langle \uparrow c, \emptyset, \uparrow c\rangle$ は，$\uparrow c$ すなわち $\lambda e.\, c \sqcup e$ という閉包演算子で表される振舞い（ストアに c を追加する振舞い）をストアに対して行い，中断せず，$c \le e$ なるストア e で終了する可能性があることを表している．後者のトレース $\langle D, \uparrow c \cap \overline{\uparrow d}, \uparrow c \cap \uparrow d\rangle$ は，D すなわち $\lambda e.\, e$ という閉包演算子で表される振舞い（ストアに制約を追加しないという振舞い）をストアに対して行い，$c \le e$ かつ $d \not\le e$ なるストア e で中断し，$c \sqcup d \le e$ なるストア e で終了する可能性があることを表している．

4. 表示的意味論

4.1 エージェントの解釈

述語記号をプロセスに変換する環境 $e : Pred \to Proc$ が与えられたとき，エージェント A の解釈 $\mathcal{G}[\![A]\!]e$ は，次のように帰納的に定義される．

$$\text{(G1)}\quad \mathcal{G}[\![\mathbf{tell}(c)]\!]e \stackrel{\text{def}}{=} c\star$$

$$\text{(G2)}\quad \mathcal{G}[\![\textstyle\sum_{i=1}^{n}\mathbf{ask}(c_i) \to A_i]\!]e \stackrel{\text{def}}{=} \prod_{i=1}^{n} c_i \to \mathcal{G}[\![A_i]\!]e$$

$$\text{(G3)}\quad \mathcal{G}[\![A_1 \parallel A_2]\!]e \stackrel{\text{def}}{=} \mathcal{G}[\![A_1]\!]e \parallel \mathcal{G}[\![A_2]\!]e$$

$$\text{(G4)}\quad \mathcal{G}[\![p(t)]\!]e \stackrel{\text{def}}{=} \{\alpha \mapsto t\}\,(e(p))$$

$$\text{(G5)}\quad \mathcal{G}[\![\exists X(A)]\!]e \stackrel{\text{def}}{=} \exists_X \mathcal{G}[\![A]\!]e$$

環境の全体集合を Env とするとき，$e \le e' \stackrel{\text{def}}{\Leftrightarrow} \forall p \in Pred(\, e(p) \subseteq e'(p)\,)$ によって $\langle Env, \le\rangle$ は完備半順序集合となり，$\lambda p.\,\mathbf{inert}$ がその最小元となる．

4.2 プログラムの意味

プログラム $Prog$ の意味 $\mathcal{P}[\![Prog]\!]$ は，環境 $\mathcal{P}[\![Prog]\!] : Pred \to Proc$ として次のように定義される．

$$\mathcal{P}_0[\![Prog]\!] \stackrel{\text{def}}{=} \lambda p.\,\mathbf{inert}$$

$$\mathcal{P}_{k+1}[\![Prog]\!] \stackrel{\text{def}}{=} \lambda p.\,\mathcal{G}[\![Prog(p)]\!](\mathcal{P}_k[\![Prog]\!])$$

$$\mathcal{P}[\![Prog]\!] \stackrel{\text{def}}{=} \lambda p.\,\bigcup_{k=0}^{\infty} \mathcal{P}_k[\![Prog]\!](p)$$

$\mathcal{G}[\![Prog(p)]\!]$ の単調性から，$\mathcal{P}[\![Prog]\!]$ は，方程式 $e = \lambda p.\,\mathcal{G}[\![Prog(p)]\!]e$ の最小不動点となる．

定義 4.1

$$\begin{aligned}\mathcal{R}(P) \stackrel{\text{def}}{=} &\{r(true)\cdot\bot \mid \langle r,s,t\rangle \in P\}\\ &\cup \{r(true)\cdot dd \mid \langle r,s,t\rangle \in P,\ r(true) \in s\}\\ &\cup \{r(true)\cdot tt \mid \langle r,s,t\rangle \in P,\ r(true) \in t\}\end{aligned}$$

定理 4.1 $\mathcal{R}(\mathcal{G}[\![A]\!](\mathcal{P}[\![Prog]\!])) = \mathcal{O}[\![A]\!]$

この定理によって，本節の表示的意味論が，操作的な意味での観測を正しく扱うことが保証される．

5. インタフェースとその利用

操作的な意味での観測を正しく扱う表示的意味論が与えられたとき，それを使用して，プロセスが特定の入力に対してどのような振舞いをするかを表現することができる．本節では，そのような定式化としてプロセスのインタフェースを導入する．その上で，前節の表示的意味論を使用することによって，最適化コンパイルを目的としたいくつかの解析の安全性を説明する．

5.1 局所的選択可能

コンパイラは，局所的選択を解消することが許されている．ここではそれを定式化するために，プロセス上に局所的選択可能という関係を定義する．

$\sum_{i=1}^{n}\mathbf{ask}(c_i) \to A_i$ という（サブエージェントを持つ）エージェント A は，ある $j, k\,(j \ne k)$ で $c_j \le c_k$ となるとき，A_k を選択しない $\sum_{i=1,\, i\ne k}^{n}\mathbf{ask}(c_i) \to A_i$ を持つエージェント A' として振舞うことができる．このとき A' は A に対して**局所的選択**を行った結果であると言う．

表示的意味論 $[\![\cdot]\!]$ が与えられたとき，上記の関係はプロセス上の関係 $[\![A]\!] \succeq_1 [\![A']\!]$ として捉えることができる．そこで $\succeq_1$ の反射推移閉包を $\succeq$ と記し，$P \succeq P'$ のとき，P は P' に**局所的選択可能**であると呼ぶ．

非決定性を持つ言語の表示的意味論では，4 節の意味論のように，プロセスの表示的意味を観測の集合によって定義することが一般的となっている．このとき，関係 $\succeq$ は半順序になる．

4 節の意味論は，計算が起こる可能性に関する方程式の最小不動点としてプロセスを定義するという意味で，最小不動点意味論と言える．このような意味論では，表示的意味によって $P \succeq P' \stackrel{\text{def}}{\Leftrightarrow} P \supseteq P'$ と定義することができる．また，局所的選択を持たない言語に対する表示的意味論を考える場合，$P \succeq P'$ は $P = P'$ として定義されるであろう．また，局所的選択に加えて発散を扱うには，上の意味で最大不動点意味論と呼べる意味論が必要となる．そのような意味論においては，P' における発散は P にも存在しているという条件が必要になる．

以下では，4 節の意味論に限定して話を進める．

例

$(true \to P \,[\!]\, c \to Q) \succeq P$ □

5.2 制約集団

制約の集合 $\{c_i\}_i$ に対して，実行時にその中の最低 1 つが満たされることを意図して $\bigvee_i c_i$ と表記したものを**制約集団**と呼び，典型的に $\vec{c}$ と記す．

制約集団はインタフェースを使った解析において，変数の動的な型情報（典型的には変数の具体化情報）を表現するために使用される．そこで以下のような略記法を用意する．変数記号 X に対して次のように定義する．

$$iX \stackrel{\text{def}}{=} \bigvee_{c\in S} c,\quad \text{where } S = \{(X = n) \mid n \in Int\} \cup \{false\}$$

ここで Int は整数を表す項の全体集合とする．簡単に言えば iX は，変数 X が整数（integer）であることを表す制約集団である．この i のような演算子を，非公式的に**型構成子**と呼ぶことにする．他にも，liX は変数 X が整数の有限リスト（list）であることを表し，vfX は X が浮動小数点数（float）の有限配列（vector）であることを表すなどとする．

D の構造によっては，無限データ構造の生成や 0 除算などによってストアが矛盾する（$false$ に過制約される）ことがある．上の定義における $false \in S$ は，そのような例外を無視するための仕掛けである．

変数記号 α は，仮引数として解釈されている．そこで α_i を α の第 i 引数（と単一化された変数記号）と見なすことにし，その上で例えば $i\alpha_1$ を $i1$ と略記する．

また，2 つの制約集団 $\bigvee_i c_i, \bigvee_j d_j$ に対して，それらの積 "$\bigvee_i c_i, \bigvee_j d_j$" を $\bigvee_{i,j}(c_i \sqcup d_j)$ によって定義する．

5.3 インタフェースの定義

プロセスは，特定の入力を仮定すると，中断せずに終了して，特定の出力をすることができることが推論できる場合がある．さらに一般化すればプロセスは，特定の入力を仮定すると，中断せずに行える特定の出力とその後の振舞いを推論できる場合がある．ここではそのような性質の定式化としてインタフェースを定義し，プロセスとインタフェースの関係を帰納的に定義する．

まずインタフェースの全体集合 $Interface$ を定義する．

$$\begin{array}{ll} Interface & E ::= P \mid \vec{c} \to B \mid E \wedge E \mid E \parallel E \\ Body & B ::= \vec{c} \gg E \mid B + B \end{array}$$

ただし P はプロセスとし，$\vec{c}$ は制約集団とする．

P は，プロセスが P のように振舞うという性質を表す．$(\bigvee_i c_i) \to B$ は，どんな c_i を満たすストアの入力を仮定してもプロセスが B のように振舞うという性質を表す．$E_1 \wedge E_2$ は，プロセスが E_1 のようにも E_2 のようにも振舞うという性質を表す．$E_1 \parallel E_2$ は，プロセスが E_1 のように振舞うプロセスと E_2 のように振舞うプロセスの並行実行のように振舞うという性質を表す．

$(\bigvee_i c_i) \gg E$ は，ある c_i を出力した後，E のように振舞うという性質を表す．$B_1 + B_2$ は，B_1 のように振舞うかまたは B_2 のように振舞うという性質を表す．

次に，プロセス P とインタフェース E の関係 $\geq \subseteq Proc \times Interface$ を，次のように帰納的に定義する．

$$\begin{array}{lll} \text{(I1)} & P \geq P & \text{iff } P \in Proc \\ \text{(I2)} & P \geq E \wedge E' & \text{iff } P \geq E,\ P \geq E' \\ \text{(I3)} & P \geq E \parallel E' & \text{iff } P = Q \parallel Q',\ Q \geq E,\ Q' \geq E' \\ \text{(I4)} & P \geq \bigvee_i c_i \to \sum_{k=1}^{n} (\bigvee_j d_j^{(k)} \gg E_k) & \\ & \quad \text{iff } \exists P'_1 \ldots P'_n (\ \forall k (P'_k \geq E_k) & \\ & \quad\ \text{and } \forall i \exists j_1 \ldots j_n \exists e_1 \ldots e_n (\ \forall k (e_k \geq d_{j_k}^{(k)}) & \\ & \quad\quad \text{and } c_i \to P \succeq \prod_{k=1}^{n} c_i \to (e_k \parallel P'_k)\ )) & \end{array}$$

インタフェースの核となる (I4) は，非決定性に対応するため難解になっている．そこで簡単のため $n = 1$ の場合を考えると，次を得る．

$$\begin{array}{lll} \text{(I4}') & P \geq \bigvee_i c_i \to \bigvee_j d_j \gg E & \\ & \quad \text{iff } \exists P' (\ P' \geq E \text{ and } \forall i \exists j \exists e (\ e \geq d_j & \\ & \quad\quad \text{and } c_i \to P \succeq c_i \to (e \parallel P')\ )) & \end{array}$$

すなわち，プロセス P は，任意の入力 c_i に対してある出力 d_j をした後，インタフェース E を持つプロセスとして振舞うことができるという性質が述べられている．

最後に，$\geq \subseteq (Interface \setminus Proc) \times Interface$ を次のように定義する．

$$\text{(I5)} \quad E \geq E' \quad \text{iff } \forall P \in Proc (\ P \geq E \ \Rightarrow\ P \geq E'\ )$$

以上によって $\geq$ は，反射律および推移律を満たすインタフェース上の関係となる．

5.4 インタフェースの例

次の 2 つのプロセスを考える．

$$\begin{array}{lll} \text{cpy} & = & (\alpha_1 \in Int) \to (\alpha_2 = \alpha_1)\star \\ \text{dbl} & = & (\alpha_2 \in Int) \to (\alpha_3 \texttt{=:=2*} \alpha_2)\star \end{array}$$

まず cpy のインタフェースを 1 つ探そう．$(\alpha_1 \in Int)$ が見えるので，入力 $i1$ を仮定する．$n \in Int$ とおくと $(\alpha_1 = n) \to \text{cpy} = (\alpha_1 = n) \to (\alpha_2 = n)\star$ となる．$(\alpha_2 = n) \in i2$ より $\text{cpy} \geq i1 \to i2 \gg \mathbf{stop}$ となる．ただし $\mathbf{stop} \stackrel{\text{def}}{=} true\star$ とする．

同様に $\text{dbl} \geq i2 \to i3 \gg \mathbf{stop}$ となる．

5.5 インタフェース上の代数計算

4 節の意味論では，インタフェースに関して以下に代表されるいくつかの公式が成り立つ．

- $(\vec{c} \to \vec{d} \gg E) \parallel E' \ \geq\ \vec{c} \to \vec{d} \gg (E \parallel E')$
- $\vec{c} \to \vec{d}, \vec{e} \gg \vec{e} \to \vec{f} \gg G \ \geq\ \vec{c} \to \vec{d}, \vec{e}, \vec{f} \gg G$

このような公式のおかげで，インタフェースを安全にボトムアップに求めることができる．

例

5.4 節の 2 つのプロセスに対して，それらの並行実行 $\text{cpy} \parallel \text{dbl}$ の持つインタフェースが次のように証明できる．

$$\begin{array}{rcl} \text{cpy} \parallel \text{dbl} & \geq & (i1 \to i2 \gg \mathbf{stop}) \parallel (i2 \to i3 \gg \mathbf{stop}) \\ & \geq & i1 \to i2 \gg (\mathbf{stop} \parallel (i2 \to i3 \gg \mathbf{stop})) \\ & = & i1 \to i2 \gg i2 \to i3 \gg \mathbf{stop} \\ & \geq & i1 \to i2, i3 \gg \mathbf{stop} \end{array}$$

□

5.6 β インタフェース

$\bigvee_i c_i \to \bigvee_j d_j \gg \mathbf{stop}$ の形のインタフェースを β インタフェースと呼ぶ．$P \geq \bigvee_i c_i \to \bigvee_j d_j \gg \mathbf{stop}$ のとき，プロセス P は，どんなストア $e \in \bigcup \uparrow c_i$ で開始されても，ある実行の方法が存在して，中断せずに終了し，ストアを $f \in \bigcup \uparrow c_i \cap \bigcup \uparrow d_j$ にすることができるということが保証される．このように，中断と終了が明示化された意味論を使い，その上で中断せずに終了できることを証明することによって，プロセスの中断しない実行手順を発見できる．

今後 "$\gg \mathbf{stop}$" は省略することがある．

ほとんどの組み込み述語は，あらかじめ β インタフェースを求めることができる．例えば，整数加算を行う 3 引数の組み込み述語 add があったとすると，$\text{add} \geq i1, i2 \to i3$ などとなる．このおかげで，データフローに従ったボトムアップな解析が可能になり，最適化された目的コードが自動的に生成できるようになる．

少し複雑にはなるが，多相述語も単一化制約を用いることによってこの枠組みでボトムアップに解析することができる．例えば，配列の長さを返す組み込み述語

vector があったとすると，任意の型構成子 τ に対して vector $\geq v\tau 1 \to i2$ などが考えられる．τ は，ボトムアップな解析に従って生成される単一化制約によって，必要に応じて i や f などの具体的な型構成子に特化され解消される．単一化ゴールの特化されたインタフェースを求めるには，これに準じた方法を用いることになる．

5.7 再帰述語の解析

$\sigma = \{\alpha \mapsto t\}$ とし，$P = b \to e \,[]\, c \to (d \,\|\, \sigma(P))$ とする．ただし $b = \overline{\exists}_\alpha b,\ c = \overline{\exists}_\alpha c$ とする．すべての中断しない線形末尾再帰述語，そしてある種の相互再帰述語の表示的意味は，この形に還元できる．ここで P は一般に α 以外の変数を局所変数に持つことに注意する．このとき $P \geq (b \to e \gg \mathbf{stop}) \wedge (c \to d \gg \sigma(P))$ となる．

線形ループとして動作することを期待して P が利用される場合，k 回反復して終了するための入力 c_k の系列 $(c_k)_k$ が原理的には計算できる．そのような系列は $c_0 \geq b,\ c_{k+1} \geq c,\ \sigma^{-1}(c_{k+1} \sqcup d) \geq c_k$ を満たす．このとき，$d_0 = \overline{\exists}_\alpha e,\ d_{k+1} = \overline{\exists}_\alpha(d \sqcup \sigma(d_k))$ とおくと，ループを反復する任意の回数 $k \geq 0$ に対して $P \geq c_k \to d_k$ が成り立つ．これらを統一的に $P \geq \bigvee_{k\geq 0} c_k \to \bigvee_{k\geq 0} d_k$ という単一の β インタフェースによって扱うことによって，さらに強力なボトムアップな解析が可能になる．

この β インタフェースは，出力を入力にフィードバックする述語呼び出しの解析には役に立たない．しかしそのような呼び出しはまれであり，その場合でも仮定が多いだけで誤りとはならない．つまりプログラム変形のときのようにデッドロックを起こす心配をする必要はない．

5.8 2 つの再帰述語の融合

並行言語ではプロセスに対して，入力と出力を入れ替えて作るプロセスを考えることができる．そしてそれは最適化コンパイルに役立てることができる．

いま，$P \geq (b \to e \gg \mathbf{stop}) \wedge (c \to d \gg \sigma(P))$ および $Q \geq \sigma^{-1}(d) \to ((b \gg \mathbf{stop}) + (c \gg \sigma(Q)))$ ただし $b = \overline{\exists}_\alpha b,\ c = \overline{\exists}_\alpha c$ および $\sigma(P) \,\|\, \sigma(Q) = \sigma(P \,\|\, Q)$ とする．最後の等式は，P と Q で使われている局所変数記号が互いに重複しないことを主張している．このとき，

$$
\begin{aligned}
P \,\|\, Q \;&\geq\; ((b \to e \gg \mathbf{stop}) \wedge (c \to d \gg \sigma(P))) \\
&\quad \| \; (\sigma^{-1}(d) \to ((b \gg \mathbf{stop}) + (c \gg \sigma(Q)))) \\
&\geq\; \sigma^{-1}(d) \;\to ((b \gg (b \to e \gg \mathbf{stop})) \\
&\qquad\qquad + \quad (c \gg \sigma(Q) \,\|\, (c \to d \gg \sigma(P)))) \\
&\geq\; \sigma^{-1}(d) \to ((b \sqcup e) + (c \sqcup d \gg \sigma(P) \,\|\, \sigma(Q))) \\
&\geq\; \sigma^{-1}(d) \to ((b \sqcup e) + (c \sqcup d \gg \sigma(P \,\|\, Q)))
\end{aligned}
$$

となる．これは，2 つのプロセスを交互に実行する場合のインタフェースとなっている．

さらにこのとき，5.7 節の再帰述語の解析の方法が適用できたとすると，2 つのプロセスは，交互に計算を進めて中断せずに終了できることが保証される．

6. クラス分けとその利用

述語間の呼び出し関係を表すグラフに基づいて述語をクラス分けすることにより，逐次的に実行できる述語のインタフェースを効率よく機械的に求めることが可能になる．ここではそのためのアルゴリズムを紹介する．

6.1 解析対象

本節では，解析対象として Flat GHC プログラムを考える．すなわち，プログラムは，$(\,p(s)$:- $G \,|\, B\,)$ という形をした**節**の集合とする．ただし G はガードゴール列，B はボディゴール列であり，各ゴールは述語呼び出しとする．述語は，組み込み述語（単一化ゴールを含む）または定義済みのユーザ定義述語とする．G が呼ぶ述語は，出力しない組み込み述語に限る．ゴールのリダクションは，G が終了する節のみを使用して行われる．

なお Flat GHC は，本質的には，2 節で定義した並行論理型言語の具体表現に過ぎないことに注意する．

6.2 クラスの定義

始めに，述語が β_k 述語および λ_k 述語であるという性質を，以下のように定義する．

定義 6.1

- β_0 述語は，組み込み述語とする．
- β_{k+1} 述語は，$\beta_0, \ldots, \beta_k$ 述語のみを呼ぶ述語とする．
- λ_0 述語は，ある k で β_k 述語となる述語とする．
- λ_{k+1} 述語は，次の性質を満たす S の最大集合の要素とする．すなわち，S に含まれる任意の述語の各定義節は，$\lambda_0, \ldots, \lambda_k$ 述語を除くと，S に含まれる述語を高々1 回しか呼ばない． □

この定義を逐次言語になぞらえて直観的に説明すると次のようになる．β_k 述語は，最大 k 回のインライン展開によって述語呼び出しが書き下せる述語を表す．また，λ_k 述語は，最大 k 重のループを形成する述語を表す．

β_0 述語は組み込み述語であるが，どの述語も呼ばないため β_{k+1} 述語でもある．また，β_k 述語は β_{k+1} 述語でもある．同様に，λ_k 述語は λ_{k+1} 述語でもある．これを踏まえて，述語の**クラス**を次のように定義する．

- クラス β_{k+1} は，β_{k+1} 述語であって β_k 述語でない述語の全体集合とする．
- クラス λ_{k+1} は，λ_{k+1} 述語であって λ_k 述語でない述語の全体集合とする．
- クラス π は，どんな k に対しても λ_k 述語とならない述語の全体集合とする．

述語のクラスは，プログラム変形によって変化することに注意する．例えば，中断しない π 述語は，CPS（Continuation Passing Style）変換 [8] に相当する変形によって λ_k 述語となる可能性を持ち，λ_k 述語はインライン展開によって β_1 述語となる可能性を持つ．

for $p \in P$ **do** $body(p) := \{\, [p_1, \ldots, p_n]$
$\quad\quad | \, (\, p(s)$:- $G \,|\, p_1(t_1), \ldots, p_n(t_n)\,)\,\}$;
for $p \in P$ **do** $dep(p) := \{s \,|\, B \in body(p),\, s \in B\}$;
for $s \in P$ **do** $callerof(s) := \{p \,|\, s \in dep(p)\}$;
for $k := 1$ **to** ∞ **do begin**
  $S := \{p \in P \,|\, dep(p) = \{\}\}$;
  if $S = \{\}$ **then break**;
  output $\beta_k \mapsto S$;
  $P := P \setminus S$;
  for $p \in P$ **do** $dep(p) := dep(p) \cap P$;
end;
for $k := 1$ **to** ∞ **do begin**
  for $p \in P$ **do** $body(p) := \{B \cap P \,|\, B \in body(p)\}$;
  $S := \{\}$;
  $S' := \{p \in P \,|\, [\_, \_|\_] \in body(p)\}$;
  while $S' \neq \{\}$ **do begin**
    $S := S \cup S'$;
    $S' := \bigcup_{s \in S'} callerof(s) \setminus S$;
  end;
  if $P \setminus S = \{\}$ **then break**;
  output $\lambda_k \mapsto P \setminus S$;
  $P := S$;
end;
output $\pi \mapsto S$;

図 2: クラス分けアルゴリズム

6.3 クラス分けアルゴリズム

図 2 のアルゴリズムは，述語の集合 P を入力とし，P に含まれる述語をクラス分けする．ただし簡単のため P に含まれる述語は，P 中の述語しか呼ばないものとする．

本アルゴリズムは効率よく実装するとプログラムの長さの二乗の時間計算量で終了することができる．

例

図 3 のプログラムに対してクラス分けを行った結果は，表 1 のようにまとめられる． □

7. 最適化コンパイルの具体例

本節ではインタフェースを求めるとき同時に逐次実行部分に対応する中間コードを生成する方法を概説する．

目的コードの最適化を可能にするため，中間コードでは原則として，コードから見えているデータは破壊的に書き換えてよいと想定する．

ヒープ上のデータの破壊的書き換えの安全性を保証するため，外から逐次実行部分を呼び出すときは，呼び出し側が破棄を許可していない入力データについてはそのコピーをもって呼び出す必要がある．並行論理プログラムでは，その読み手が破棄してよいデータを見つけるための方法として参照数解析 [6] などが存在しており，データの破棄を許可するために利用することができる．

```
treesum(T,Res) :- treesum2(T,0,Res).
treesum2(leaf(N),    S0,S) :- add(S0,N,S).
treesum2(node(T1,T2),S0,S) :-
    treesum2(T1,S0,S1), treesum2(T2,S1,S).
listsum(L,Res) :- listsum2(L,0,Res).
listsum2([],    S0,S) :- S=S0.
listsum2([N|Ns],S0,S) :-
    add(S0,N,S1), listsum2(Ns,S1,S).
add(X,0,   Res) :- Res=X.
add(X,s(Y),Res) :- inc(X,X1), add(X1,Y,Res).
inc(X,X1) :- X1=s(X).
inc2(X,X2) :- X2=s(s(X)).
inc2(X,X2) :- inc(X,X1), inc(X1,X2).
```

図 3: クラス分け対象のプログラム

クラス	述語
β_1	inc
β_2	inc2
λ_1	add
λ_2	listsum, listsum2
π	treesum, treesum2

表 1: 図 3 のクラス分けの結果

なお，本節で使用する中間コードは説明のために架空に作られたものであり，まだ形式的な定義は存在しない．また，原理的に実現可能であることを示唆しただけであり，具体的な実装法はこれから設計しなければならない．

7.1 ループ最適化実行

配列を扱う典型的なループの最適化実行の例を示す．配列隣接要素の平均をとる図 4 のプログラムを考える．

始めに，述語のクラス分けを行う．blur は λ_1 である．

blur に対して，インタフェースを求める．節 c_1 より

- blur $\geq i1, i2, (\alpha_1$ `>=` $\alpha_2) \rightarrow (\alpha_3$ `=` $\alpha_4)$

を得る．次に節 c_2 である．λ_1 以外の述語のゴールのインタフェースを求めると，次のようになる☆．

- $(K$ `:=` $K0$ `+1`$) \geq iK0 \rightarrow iK$
- vector_element$(V0, K0, A) \geq v\tau_1 V0, iK0 \rightarrow \tau_1 A$
- vector_element$(V0, K, B) \geq v\tau_2 V0, iK \rightarrow \tau_2 B$
- $(C$ `$:=(` A `+` B `)/2`$) \geq fA, fB \rightarrow fC$
- set_vector_element$(V0, K0, C, V1)$
  $\geq v\tau_3 V0, iK0, \tau_3 C \rightarrow v\tau_3 V1$

ここで τ_i は型構成子上を動くメタ変数とする．

次に入力 $i1, i2, (\alpha_1 < \alpha_2)$ を仮定して，上記 5 ゴールの並行実行 P に対して単一の β インタフェースを作る．ここで各変数記号に対してその型構成子は唯一とするために制約 $\tau_1 = \tau_2 = \tau_3 = f$ を生成すると，P は上記の

☆ 本節では $\{\alpha \mapsto \langle t_1, t_2 \rangle\}(P)$ を $P(t_1, t_2)$ などと略記する．

```
c1 :: blur(K0,N,V0,V) :- K0 >=N | V=V0.
c2 :: blur(K0,N,V0,V) :- K0 < N | K:=K0+1,
      vector_element(V0,K0,A),
      vector_element(V0,K,B),
      C $:= (A+B)/2,
      set_vector_element(V0,K0,C,V1),
      blur(K,N,V1,V).
```

図 4: ループ最適化実行ができるプログラム

```
blur_i1i2vf3:
  if (a1>=a2) {
    tell a4,a3
    return null
  }
  if (a1<a2) {
    k = a1 + 1
    a = a3[a1]
    b = a3[k]
    c = (a+b)/2
    v1 = a3          (⋆)
    v1[a1] = c       (⋆)
    a1 = k
    a3 = v1
    return blur_i1i2vf3
  }
```

(a) 最適化前

```
blur_i1i2vf3_vf4:
  while (a1<a2) {
    k = a1 + 1
    a = a3[a1]
    b = a3[k]
    c = (a+b)/2
    a3[a1] = c
    a1 = k
  }
  tell a4,a3
  return null
```

(b) 最適化後

図 5: 図 4 から生成される中間コード

順に中断せず終了可能と分かり，次の関係が求まる．

- $P \geq i1, i2, vf3 \rightarrow iK, fA, fB, fC, vfV1$

この結果，節 c_2 全体として次のインタフェースを得る．

- blur $\geq$ $i1, i2, (\alpha_1 \texttt{<} \alpha_2), vf3 \rightarrow iK, vfV1,$
  $(K \texttt{=:=} \alpha_1 \texttt{+1}) \gg \mathrm{blur}(K, \alpha_2, V1, \alpha_4)$

この結果に合わせて，再び節 c_1 について次を得る．

- blur $\geq i1, i2, (\alpha_1 \texttt{>=} \alpha_2), vf3 \rightarrow vf4$

最後の 2 つの関係を得る過程で，ボディゴールの中断しない実行順序も見つかっているので，同時に図 5 (a) のような中間コードが得られる．

中間コードは，a1, a2 などを暗黙の引数レジスタとして持っている．x = y は，x の値を y の値で上書きする代入文である．一方 tell x,y は，x が指すメモリ番地に y の値を書き込む出力文である．また return は，コンテキストスイッチが可能な goto 文に相当する．

図 5 (a) の中間コードは，set_vector_element が提供した 2 行 (⋆) において配列のコピーを行っていることに注意する．このコピーは，中間コードに対する最適化によって簡単に除去できるはずである．

最後の 2 つのインタフェースを受けて，ループ誘導変数の検出と無発散の証明に成功すれば，次が得られる．

- blur $\geq i1, i2, vf3 \rightarrow vf4$

```
intlist_i1i2_si3:
  if (a1>=a2) {
    tell a3,[]
    return null
  }
  if (a1<a2) {
    tell a3,[_|_]
    tell a3[1],a1
    k = a1 + 1
    a1 = k
    a3 = a3[2]
    return
      intlist_i1i2_si3
  }
```

```
sum_si1i2:
  if (a1=[]) {
    tell a3,a2
    return null
  }
  if (a1=[_|_]) {
    x1 = a2 + a1[1]
    a1 = a1[2]
    a2 = x1
    return sum_si1i2
  }
```

図 6: 図 1 から生成される中間コード

```
P_i1i2i4_si3:
  if (a1>=a2) {
    tell a3,[]
    tell a5,a4
    return null
  }
  if (a1<a2) {
    tell a3,[_|_]
    tell a3[1],a1
    k = a1 + 1
    a1 = k
    x1 = a4 + a3[1]
    a3 = a3[2]
    a4 = x1
    return P_i1i2i4_si3
  }
```

図 7: 図 6 を融合して得られる中間コード

このインタフェースに対する最適化された中間コードとして，最終的に図 5 (b) のようなものが得られる☆．

7.2 2 つの述語の融合

ループ融合 [4] が可能な，冒頭で示した図 1 のプログラムを最適化コンパイルする例を示す．

始めに，述語のクラス分けを行う．sum, intlist は λ_1 であり，stairs は λ_2 である．

つぎに λ_1, λ_2 の順に逐次実行部分を抽出する．

sum, intlist に対してインタフェースを求めると

- sum $\geq$ $n1, i2 \rightarrow i3 \gg \mathbf{stop}$
  $\wedge$ $c1, i\alpha_{11}, i2 \rightarrow iX1 \gg \mathrm{sum}(\alpha_{12}, X1, \alpha_3)$
- intlist $\geq$ $i1, i2, (\alpha_1 \texttt{>=} \alpha_2) \rightarrow n3 \gg \mathbf{stop}$
  $\wedge$ $i1, i2, (\alpha_1 \texttt{<} \alpha_2) \rightarrow c3, i\alpha_{31}, iK,$
  $(K \texttt{=:=} \alpha_1 \texttt{+1}) \gg \mathrm{intlist}(K, \alpha_2, \alpha_{32})$

を得る．ただし n は nil，c は cons をそれぞれ表す型構成子とし，α_{12} は α_1 の第 2 引数を表す変数記号などとする．このとき，同時に図 6 のような中間コードを得る．

☆ 処理系にいわゆる fairness を要求しない立場を取る場合，ループ誘導変数の検出や無発散の証明をすることなく，前者の中間コードから直接後者の中間コードにコンパイルできることに注意する．

なお si は，整数のストリーム（stream）を意図しており，car 部が整数の cons または nil を表している．

次に stairs に移る．$\mathrm{intlist}(K0, N, S) \parallel \mathrm{sum}(S, X0, X) = P(K0, N, S, X0, X)$ とするとき，上記の結果から

- $P \geq i1, i2, (\alpha_1 \texttt{>=} \alpha_1), i4 \rightarrow n3, i5 \gg \mathbf{stop}$
  $\wedge\ i1, i2, (\alpha_1 \texttt{<} \alpha_2), i4 \rightarrow c3, i\alpha_{31}, iK, iX1,$
  $(K \texttt{:=} \alpha_1 \texttt{+1}) \gg P(K, \alpha_2, \alpha_{32}, X1, \alpha_5)$

となる．このとき同時に，上記の中間コードを組み合わせて，図 7 のような中間コードが合成できるであろう．

本節の方法によって，プロセスの動的な生成が必要となる直前までが逐次実行部分として抽出されるはずである．したがって最終的に処理系を完成させるには，逐次実行部分を実行するクロージャを効率よく管理する実行時処理系を提供すればよいと考えられる．

8. 関 連 研 究

細粒度スレッドが利用できる並列言語の最適化コンパイラの関連研究としては 9)，11) などがある．

9) は Scheme を拡張した並列オブジェクト指向言語 Schematic の最適化コンパイラの構成法について論じている．とくに中間言語から逐次の目的コードを生成する方法は参考になる．しかし並行論理型言語のようにプロセス間通信やデータ構造に自由度のある計算モデルに対して適用する場合，最適化を効率よく行うには事前にある程度まで逐次実行部分を抽出して逐次化しておく必要がある．そのためには制御構造に注目する必要があるのだが，従来は並行論理型言語に対してそのようなモジュール性の高い最適化は行われてこなかった．

11) は並列論理型言語 Fleng の最適化コンパイラに関するもので，最適化を行う要素技術が参考になる．ただし最適化がソースコードを対象にした解析で行われるため，最適化のモジュール性が低いという欠点があった．

不規則な並列性を持ったプロセスの実装法の研究としては 10) などがある．そこでは，とくに効率のよいスレッド管理および負荷分散の方法が参考になる．

並行論理型言語の表示的意味論の重要な研究としては 1)〜3) などがある．中でも 3) は，並行論理プログラムを一般化した並行制約プログラムに対する表示的意味論のおそらく最初の研究であり，非決定的プロセスのストアに対する振舞いが，トレース演算子の集合として表現できることを明らかにしたという意味で特に重要である．ところで 3) は，制約追加の失敗時にカオスとなる計算モデルに対して，インタラクションの存在に関する最大不動点意味論を定義している．そこでは，プロセスの表示的意味が，単一のインタラクションを表す（一般には定義域の異なる）トレース演算子の集合として定義されていたため，表示的意味が煩雑で扱いにくく，様々な証明を行うには向いていなかった．それに対して本論文では，制約追加の失敗検出をしない計算モデルを対象にしたことによって，インタラクションが作る（可算な）系列を表す（定義域の等しい）トレース演算子を基本とした簡素な表示的意味を定義することが可能になり，それによって簡潔な証明を可能にした．2) は，操作的アプローチでありながら極限におけるストアを得るための表示的意味論を深く論じている．ただし言語に制限を設けているため，理論の応用範囲は限られている．

9. まとめと今後の課題

本論文で我々は，並行論理プログラムに対して，逐次的に実行できる部分を抽出する枠組みと，その安全性に対する保証を与えた．提案した枠組みは，並行プロセスの動作に関する期待の定式化であるインタフェースを利用したボトムアップな解析に基づいている．とくに，ボトムアップな解析の手順がそのまま中間コードの生成手順にもなっているという点が，従来の最適化コンパイラの研究とは異なった考え方であり，しかも実用の観点からも注目すべきことであると思われる．

理論面では，プロセスが持つインタフェースのような性質を，表示的意味論を使って定式化する枠組みを与えた．それによって，様々な応用に対して意味論に支えられた定式化を行うことが可能となった．実際，本論文で示した最適化コンパイラの構成法には，適用できる最適化要素技術の範囲が広いという特徴がある．

また，本論文で示したインタフェースの定式化の枠組みは，並行論理プログラムのみならず，一般の並行制約プログラムにおいても広く利用できるものと期待される．

意味論自身については，様々な証明を簡潔に行うため，最小不動点に基づく簡素な表示的意味論を与えた．とくに，トレース演算子を使いながら中断や終了を扱うという点が新しい．また，述語呼び出し時の変数名の付け替えを定式化するために変数変換という記法を定義した．変数変換は，本論文の随所で重要な役割を担っている．

今後の課題として，最大不動点意味論と最小不動点意味論を組み合わせることによって，再び一貫した表示的意味論が得られることを示すことが挙げられる．そのような意味論を使うと，必然性を表現するインタフェースを定義することが可能になる．それによって仕様によるプロセス合成を定式化することや，新しい意味論をプログラム変形の証明に使用することが考えられる．

一方，7 節で使用した中間コードを形式的に定義し，Flat GHC に基づく言語である KL1 のプログラムに対して実際にループ最適化実行などを行う最適化コンパイラを試作し評価することも重要であろう．

謝辞

本研究の一部は，科学研究費補助金 (C)(2)11680370 の助成を得て行われた．

参 考 文 献

1) F.S. de Boer and C. Palamidessi: On the semantics of concurrent constraint programming. Invited paper in *Proc. of ALPUK 92*, Workshops in Computing, pp. 145–173, Springer-Verlag, 1992.
2) F.S. de Boer, A. Di Pierro, and C. Palamidessi: Nondeterminism and Infinite Computations in Constraint Programming. International Workshop on Topology and Completion in Semantics, Chartres, 1993.
3) Saraswat, V. A., Rinard, M. C. and Panangaden, P.: Semantic Foundations of Concurrent Constraint Programming, *Conf. Record of the Eighteenth Annual ACM Symp. on Principles of Programming Languages*, ACM Press, pp. 333–352, 1991.
4) Ueda, K. and Furukawa, K.: Transformation Rules for GHC Programs, *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, ICOT, Tokyo, pp. 582–591, 1988.
5) Ueda, K. and Morita, M.: Moded Flat GHC and Its Message Oriented Implementation Technique, *New Generation Computing*, Vol. 11, No. 3–4, pp. 3–43, 1993.
6) Ueda, K.: Linearity Analysis of Concurrent Logic Programs. In *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, pp.253–270, 2000.
7) 加藤紀夫, 上田和紀: 並行論理型言語における同期ポイントの移動の安全性について, 情報処理学会論文誌: プログラミング, Vol.41, No.SIG 2 (PRO 6), pp. 13–28, 2000.
8) Andrew W. Appel: Compiling with Continuations, Cambridge University Press, 1992.
9) 大山恵弘, 田浦健次朗, 米澤明憲: 動的なスレッド生成をサポートする言語のコンパイル技法, SWoPP'96 予稿集, 1996.
10) 大山恵弘, 田浦健次朗, 米澤明憲: 細粒度スレッド生成をサポートする言語の共有メモリ並列計算機上での実装とその性能評価, プログラミングと応用のシステムに関するワークショップ予稿集, 1998.
11) 荒木拓也: 並列論理型言語 Fleng の最適化コンパイラに関する研究, 東京大学大学院工学系研究科 博士論文, 1999.