並行論理プログラミングにおける静的モード体系の応用的側面に関する研究

(課題番号 07680341)

平成7年度～平成9年度科学研究費補助金
(基盤研究 (C)(2))
研究成果報告書

平成11年3月

研究代表者

上田 和紀

(早稲田大学理工学部教授)

はじめに

並行論理プログラミング言語は，論理プログラミング・パラダイムに情報の流れを制御するための同期機構を導入し，並行処理 (concurrency) 記述のための言語としたものである．研究代表者の提案した並行論理プログラミング言語 GHC (Guarded Horn Clauses) は，意味論的に非常に簡潔な通信・同期機構をもちながら，構成が動的に変化する並行プロセス系の通信プロトコルを自然に記述できる柔軟性も併せもつ．

並行論理プログラミング言語の実用化を図るには，プログラミング支援と処理効率の両面から，プログラムの静的解析技法がきわめて重要となる．研究代表者はこれまでに，GHC プログラムの情報の流れ (モード) を論じるためのモード体系と，静的モード解析技法を提案し，その理論的側面を整備してきた．GHC に静的モード体系を導入することにより，共有単一代入変数を用いたプロセス間通信の送受信者の区別や受信者数を解析することが可能になり，また共有単一代入変数による通信の失敗が起きないことを保証できるようになった．

本研究では，この静的モード体系に対し，新たに応用的側面から検討を加え，その有用性をさまざまな角度から実証してきた．

1. モード解析システムの実装 — 並行論理型言語 KL1 のためのモード解析システム klint を，KL1 言語自身で実装した．現実のプログラムに適用することによってシステムの評価・拡充を繰返し，複雑な KL1 プログラムに対応できる実用的なシステムとした．
2. モード体系の下での記述能力の検証 — klint システム自身をはじめとする実用規模の並行論理プログラムの解析を通じ，静的モード体系を備えた並行論理型言語の記述力が十分であることを明らかにした．
3. デバッグへの応用 — モード体系はプログラムの誤りの静的検出にも非常に有効である．そこで，モードづけできないプログラムの誤りを効率よく解析するアルゴリズムを提案し，実装を行なった．本技法は制約に基づく静的体系一般に適用可能なものとなった．さらにプログラム誤りの自動修正の方法を提案し，その修正能力を確認した．
4. 静的モード体系を利用した処理系の改善 — モード解析情報に，データ型やデータ参照数についての解析情報を併用することにより，タグ判別をはじめとする動的な判断の多くが省略でき，手続き型言語で直接記述した場合と大きく変わらない性能が得られることを明らかにした．

静的モード解析に関する過去の関連研究は，ほとんどが抽象解釈技術に基づくものであった．これに対し本研究は，制約 (constraints) 概念に基づく独自のモード体系をベー

スにしている．本モード体系は，概念的に簡潔であるばかりでなく，大域的な解析も，ほとんどの場合ユニフィケーション問題として効率良く実行することができる．その実用化は，プログラミングと処理系作成の両面に大きな波及効果をもつと期待される．また，静的な型体系を備えたプログラム言語は数多いが，静的なモード体系は，Moded Flat GHC独自のものである．したがってその実用的意義を実証的に調べることはきわめて重要である．

本成果報告書では，このモード体系の応用的側面に関する研究成果について，学会誌や国際会議の発表論文等に基づき詳述する．

研究組織

研究代表者: 上田 和紀 (早稲田大学理工学部教授)

(研究協力者: 長 健太 (早稲田大学大学院理工学研究科，現在 (株) 東芝))

(研究協力者: 土山 了士 (早稲田大学大学院理工学研究科，現在三菱電機 (株)))

(研究協力者: 網代 育大 (早稲田大学大学院理工学研究科))

研究経費

平成 7 年度	1,300 千円
平成 8 年度	800 千円
平成 9 年度	300 千円
計	2,400 千円

研究発表

- 学会誌等 (査読論文)

[1] Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. International Workshop on Parallel Symbolic Languages and Systems*, Lecture Notes in Computer Science 1068, Springer-Verlag, Berlin, pp. 134–153, April 1996.

[2] 長健太，上田和紀: モード誤りをもつ並行論理プログラムの静的デバッグ手法，1996 年並列処理シンポジウム論文集，情報処理学会，pp. 219–226, 1996 年 6 月．

[3] Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs. In *Proc. 1996 Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, M. Maher (ed.), The MIT Press, September 1996, pp. 215–229.

[4] 網代育大，長健太，上田和紀：静的解析と制約充足によるプログラム自動デバッグ．コンピュータソフトウェア, Vol. 15, No. 1 (1998), pp. 54–58.

[5] Ajiro, Y., Ueda, K. and Cho, K., Error-Correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP'98)*, LNCS 1520, Springer, October 1998, pp. 40–54.

[6] Ueda, K. and Tsuchiyama, R., Optimizing KLIC Generic Objects by Static Analysis. In *Proc. 11th Int. Conf. on Applications of Prolog*, Prolog Association of Japan, September 1998, pp. 27–33.

- 口頭発表

[7] Strong Moding in Concurrent Logic/Constraint Programming. Advanced tutorial given at the Twelfth International Conference on Logic Programming, Kanagawa, Japan, June 1995.

[8] 長健太，上田和紀：並行論理プログラムのモード解析手法を用いた静的診断，1996 年度人工知能学会全国大会 (第 10 回) 論文集，人工知能学会，pp. 25–28, 1996 年 6 月．

[9] 長健太, 上田和紀：制約概念に基づくプログラム解析・診断・デバッグ－並行論理型言語への適用，日本ソフトウェア科学会第 13 回大会論文集，日本ソフトウェア科学会，pp. 37–40, 1996 年 9 月．

[10] Kazunori Ueda, Diagnosis of Concurrent Logic Programs. Dagstuhl Seminar (9704) on High-Level Concurrent Languages, Schloss Dagstuhl, Germany, January 1997.

[11] 網代育大，長健太，上田和紀，制約に基づく解析による並行論理プログラムの自動デバッグ．1997 年度人工知能学会全国大会 (第 11 回) 論文集，pp. 205–208, 1997 年 6 月．

[12] 網代育大，長健太，上田和紀，静的解析と制約充足によるプログラム自動デバッグ．日本ソフトウェア科学会第 14 回大会論文集，pp. 533–536, 1997 年 10 月．

[13] Kazunori Ueda, Constraint-based Automated Debugging. Dagstuhl Seminar (9741) on Concurrent Constraint Programming: The Next Ten Years, Schloss Dagstuhl, Germany, October 1997.

[14] 上田和紀：並行論理プログラムの参照数解析，情報処理学会第 20 回プログラミング研究会 (SWoPP'98)，1998 年 8 月．

[15] 網代育大，上田和紀：並行論理プログラム静的解析系 Kima の実装．情報処理学会第 58 回全国大会 4N-04, 1999 年 3 月．

本報告書の各章の構成を簡単に記す．各章ができるだけ独立に読めるようにするため，基本的な定義等は繰返し掲げている．

第 1 章は，本研究の基礎となった並行論理プログラミング言語のモード体系について詳述している．また KL1 という現実の並行論理プログラミング言語の諸機能の取り扱い

について詳細に検討している．本章の内容は，本研究に先立って行なわれたモード体系の研究成果 (巻末文献 [46]) を大幅に拡張発展させたものである)．

第 2 章は，KL1 のためのモード解析系の実装と，現実の KL1 プログラムへの適用経験，およびモード体系の拡張について議論している．本章の内容は前掲の論文 [1] に基づいている．

第 3 章は，モード体系をプログラムの誤りの静的検出に適用する技法について論じている．本章の内容は前掲の論文 [2] および [3] に基づいている．

第 4 章は，モード体系をプログラムの誤りの自動修正に適用する技法について論じている．本章の内容は前掲の論文 [4] および [5] に基づいている．

第 5 章は，モード体系を他の静的解析技法とともに用いて，処理系最適化を行なう方法について論じている．本章では特に，KLIC 処理系における汎用オブジェクトの最適化に焦点を当てている．本章の内容は前掲の論文 [6] に基づいている．

第 6 章は，モード体系と同様の技法で定式化した参照数解析のための体系を紹介し，その性質や応用を論じている．本章の内容は前掲の発表 [14] の内容に基づいている．

目 次

第1章　並行論理プログラミング言語のモード体系

1.1　はじめに

第五世代コンピュータの研究基盤化プロジェクトにおいて，汎用計算機上の KL1 処理系 (KLIC) の開発が進められてきた [5]．汎用機上の KL1 処理系を，使いやすく，しかも他言語に伍する効率を備えたものにするには，コンパイル時のプログラム解析技法と，プログラム解析結果に基づく最適化技法がきわめて重要となる．そこで本調査研究では，KL1 プログラムのデータの流れ (モード) に関する解析技法を中心に，関連諸側面の検討をすすめ，最終的にはその成果を KL1 処理系に役立てることを目標とする．

1.1.1　研究方法

本研究の基礎となるのは，KL1 の基礎をなす Flat GHC のモード解析技法である．そこで，まずこのモード解析技法が，(1) どのように定式化できるか，(2) 理論的に何を保証するのか，(3) どのくらいの効率で解析できるのか，についての検討を，それぞれ進めることとする．

(1) については，入出力モードの体系はすでに提案されているが，プログラムが，その入出力モードに与える構文的制約規則は研究途上であった．この規則の十分簡潔な表現を得ることを目指す．また，モードに関する制約充足問題を解く実用的な方法を明らかにする．(2) については，モード解析の基本定理の形で，解析が成功したプログラムの性質を表現し，証明する．(3) については，モード解析の計算量についての考察を進める．

さらに，多くの現存する KL1 プログラム解析のためには，KL1 にはあるが Flat GHC にはない種々の機能をどう扱うかを検討する必要がある．その第一段階として，種々の組込述語を，Moded Flat GHC の枠組でうまく扱うことができるかを検討する．

1.1.2　成果概要

KL1 の基礎をなす Flat GHC のモード解析については，これまでの検討によって，下記のような構成をもつ解析システムを設定することができた:

(a) 入出力モードの体系
(b) プログラムやゴール節が，その入出力モードに与える構文的制約
(c) (b) の制約を解くためのアルゴリズム

(a) は，プログラムやゴール節中の述語の引数がとりうるデータ構造の各箇所が，入力と出力のいずれであるかを区別するためのものである．モード体系の設計にあたっては，双方向通信やストリームのストリームのような，並行論理型言語特有の柔軟な通信が表現できるということに留意した．

(b) は，与えられたプログラムやゴール節の入出力モードを定める制約条件を，論理式の形で列挙したものである．これらの制約条件は，プログラムやゴール節を構成する各記号に付随して与えられ，それらの制約をすべて満たすものが正しい入出力モードである．正しい入出力モードが存在するプログラムやゴール節を，well-moded であるという．

(c) は，(b) が与える制約の集合（つまり連立方程式）を解くアルゴリズムである．簡単なプログラムに対しては，ループを持つ素性構造 (feature structure) のユニフィケーションと同じ方法で解けることがわかった．

このモード体系を備えた Flat GHC を，Moded Flat GHC という．Moded Flat GHC のプログラムは，一定の条件下で，(1) ゴール節のリダクションが well-modedness を保存する，(2) 実行が終了すると，最初のゴール節中のすべての変数が，基底項に具体化する，という二つの強い性質を満たすことが証明できた．(1) は，モード解析の基本定理とも呼ぶべきものである．また，(1) の系として，well-moded なゴール節では，ユニフィケーションが，常に未定義変数と項とのユニフィケーション，つまり単なる代入であることが静的に保証される．

このことが保証できると，ユニフィケーションの実装が大幅に簡潔化，効率化でき，特に KL1 の分散実装に大きなメリットがあると期待される．またモード解析は，多くの言語に見られる型解析と同様，プログラムの誤りを静的に発見するのにも極めて有用であることが判明してきた．

本解析手法の大きな特徴は，プログラム全体を一度に解析せずに，分割して解析しても，解析の効率があまり低下しない点である．モード解析の実用的な手間を評価したところ，一括解析の場合，プログラムの大きさ n と，使用するデータ構造の複雑さとにほとんど比例する程度に押さえることができ，分割解析しても全体で $O(\log n)$ 倍悪化するだけであることがわかった．

本解析技法を KL1 プログラムに適用するには，ベクタ等の KL1 特有のデータ型や，純粋な Flat GHC の枠組では説明できない組込述語の扱いを検討しなければならない．本年度は，それらについての基礎検討を行なった結果，さらに検討の余地のある高階機能を除けば，基本的には KL1 特有の機能を用いたプログラムも，問題なくモード解析ができる (あるいは，解析できるように容易にプログラムを書き換えることができる) ことが判明した．ただし，現在の KL1 の組込述語の一部については，モードの観点から仕様の変更が望ましいことなども明らかとなった．

1.2 Flat GHC

本節では，モード体系の設計のベースとなった Flat GHC の概要を述べる [43]．伝統的には，組込述語の呼出しのみをガードゴールとして許したものが Flat GHC と呼ばれることが多いが，後者の定義は，組込述語の集合を定義しないと完全にならない上に，組込述語の選択によってプログラムの性質が変わることになり，理論上も処理系作成上も

問題が多い．

なお本報告は，並行論理型プログラミングおよびGHCとKL1の概要に関する予備知識を前提としている．これらについては，[26, 27, 32, 33, 34, 35, 42]等を参照してほしい．

1.2.1 Flat GHCの定義

Flat GHCは，多くの概念を論理および論理プログラミングから借用している．ここでも，下記の概念は既知とする[17]:

- 変数(記号)，関数(記号)，定数(記号)(0引数関数記号とみなす)，述語(記号)
- 項，アトム(原子論理式)
- 代入(後置演算子として表記)，名前替え(renaming)
- 単一化，mgu (most general unifier)

項t_1とt_2に対して，$t_1\theta \equiv t_2$('$\equiv$'は構文的同一性を表わす)であるような代入θがあるとき，t_1がt_2に**マッチする**，という．

Flat GHCのプログラムは，ガードつき節の集合である．**ガードつき節**(以下では**(プログラム)節**ともいう)とは，hをアトム，GとBをアトムのマルチ集合とすると，

$$h :- G \mid B$$

という形のものである．hを節の**頭部**，G中のアトムを**ガードゴール**，B中のアトムを**ボディゴール**という．**コミット演算子**'|'より前の部分をプログラム節の**ガード**，後の部分を**ボディ**という．

空のボディを持つプログラム節を，**単位節**と呼ぶ．プログラム中で，頭部に同一の述語記号pをもつすべての節の集合を，pの**定義** (procedure)という．述語記号pをもつ(ガードまたはボディ)ゴールは，pを**呼び出す**という．

直観的には，個々のガードつき節は，ゴールの書換え規則である．ここで，

- hは，書き換えるべきゴール(bとする)にマッチすべきテンプレート，
- Gは書換えのための付帯条件(b中の変数の値を制約せずに実行できなければならない)，
- Bは，bにとって代わるべき(サブ)ゴールのマルチ集合

である．プログラムが節の**集合**であるということは，節の順序に意味がないばかりでなく，同じ節(変数を名前替えしただけのものは同一とみなす)が複数回出現することにも意味がないことを示す．一方，GおよびBは**マルチ集合**である．これは，非決定性(indeterminacy)のために，構文的に同一の二つのゴールが異なるふるまいを示すことがあるからである．

プログラムを起動するには，

$$:- B$$

という形のゴール節を使う．これは，書き換えてゆくボディゴールの最初のマルチ集合を指定するものである．

個々のゴールは，$t_1 = t_2$ の形をした単一化ゴールか，非単一化ゴール ('=' 以外の述語を呼び出すゴール) かのいずれかである．単一化ゴールは，代入 (空かも知れない) を生成することで，変数のとりうる値を制約する．非単一化ゴールは，代入 (空かも知れない) を観測後，他のゴールに書き換えられてゆく．ガードつき節のガードは，書換えの前にどのような代入を観測しなければならないかを指定しており，Flat GHC に同期メカニズムを与えている．

Full GHC では，ガードゴールとしてどんなアトムが現れてもよかった．しかし，これは並行言語としては不必要に強力であることがわかった．Flat GHC では，ガードゴールは，書換えのための条件を指定するものであるので，まずはその目的に合致するテスト述語というクラスを定義する．述語 p がテスト述語であるとは，p が単位節の集合によって定義されているということである．テスト述語を呼び出すゴールは，成功するか否かだけが関心事であり，ゴールの外から観測可能な代入を生成しないという性質を持つ．もう一つの重要な性質は，その決定性である．つまり，成功するか否かが，現在の (変数の束縛) 環境から一意に定まる．

Flat GHC プログラムは，フラットなガードつき節の集合である．ここで，フラットなガードつき節とは，ガードゴールを，単一化ゴールとテスト述語の呼出しとに制限した節のことである．

なお，具体的なプログラムの記述においては，アトムの空でないマルチ集合は，要素アトムを ',' (カンマ) で区切って並べて表記する．また，アトムの空のマルチ集合は `true` と表記する．節の集合は，'.' (ピリオド) で終る節の並びで示す．

1.2.2 Flat GHC の操作的意味

ここでは，Flat GHC の操作的意味を，今や標準的手法となった Plotkin のスタイル [22] によって定義する．本節では，Flat GHC のゴールが観測・生成する情報や，変数の束縛環境を，代入としてではなく，より一般的な等号制約とみなすことにする．これによって，単一化の失敗の扱い等が統一的になる．

B をゴールのマルチ集合とし，C を (等号制約を表現する) 等式のマルチ集合とする．また V を変数の集合とする．$\mathcal{V}_F$ で，構文要素 F の中に出現するすべての変数の集合を表わす．

Flat GHC における計算のコンフィギュレーションとは，$\langle B, C\rangle : V$ と書く三つ組であり，$\mathcal{V}_B \cup \mathcal{V}_C \subseteq V$ を満たすようなものである．これは，書き換えるべきゴール，現在の束縛環境，および現在の計算にすでに使用した変数の集合を記録したものである．

プログラム $\mathcal{P}$ の下での計算は，初期コンフィギュレーション $\langle B_0, \emptyset\rangle : \mathcal{V}_{B_0}$ から始まる．ここで B_0 は，与えられたゴール節のボディである．

以下で定めようとしているのは，遷移関係 $c_1 \longrightarrow c_2$，つまり「コンフィギュレーション c_1 は c_2 に書き換えることができる」という関係である．使用するプログラム $\mathcal{P}$ を陽に示す必要のあるときは，$\mathcal{P} \vdash c_1 \longrightarrow c_2$ という形を用いる．こちらは,「プログラム $\mathcal{P}$ の下で，c_1 は c_2 に書き換えてよい」と読む．記号 $\xrightarrow{*}$ は，$\longrightarrow$ の反射推移閉包を表わす．

- $\forall.(\neg(f(\mathtt{X}_1,\ldots,\mathtt{X}_m)=g(\mathtt{Y}_1,\ldots,\mathtt{Y}_n)))$, 異なる関数 f, g の対それぞれについて
- $\forall.(\neg(t=\mathtt{X}))$, $\mathtt{X}$ を含む，$\mathtt{X}$ 以外のすべての項 t について
- $\forall.(\mathtt{X}=\mathtt{X})$
- $\forall.(f(\mathtt{X}_1,\ldots,\mathtt{X}_m)=f(\mathtt{Y}_1,\ldots,\mathtt{Y}_m)\Rightarrow\bigwedge_{i=1}^{m}(\mathtt{X}_i=\mathtt{Y}_i))$, 各 m 引数関数 f について
- $\forall.(\bigwedge_{i=1}^{m}(\mathtt{X}_i=\mathtt{Y}_i)\Rightarrow f(\mathtt{X}_1,\ldots,\mathtt{X}_m)=f(\mathtt{Y}_1,\ldots,\mathtt{Y}_m))$, 各 m 引数関数 f について
- $\forall.(\mathtt{X}=\mathtt{Y}\Rightarrow\mathtt{Y}=\mathtt{X})$
- $\forall.(\mathtt{X}=\mathtt{Y}\wedge\mathtt{Y}=\mathtt{Z}\Rightarrow\mathtt{X}=\mathtt{Z})$

図 1.1: 節形式で表わした Clark の等号理論 $\mathcal{E}$

規則

$$\frac{\mathcal{P}_1\vdash t_1}{\mathcal{P}_2\vdash t_2}\quad(\text{if } \mathit{Cond})$$

は，もし $\mathcal{P}_1$ の下で遷移 t_1 が許され，付帯条件 Cond が成り立つならば，$\mathcal{P}_2$ の下で遷移 t_2 が許されると読む．前提部と付帯条件は，空ならば省略してよい．

Flat GHC の遷移規則は以下の三つである．ここで，$F\models G$ は，G が F の論理的帰結であることを表わす．また，$\forall\mathcal{V}_F.F$ と $\exists\mathcal{V}_F.F$ は，それぞれ $\forall.F$ および $\exists.F$ と略記する．さらに，[24] にならい，$\delta V.F$ で $\exists(\mathcal{V}_F\backslash V).F$ を表わすこととする．ここで V は変数の有限集合である．また我々は，述語の集合から関数の集合への単射 '$\mp$' があるものと仮定する．この単射は，アトムの集合から項の集合へ自然に拡張できる．$\mathcal{E}$ で，Clark の等式理論 (図 1.1) を表わす．

$$\frac{\mathcal{P}\vdash\langle B_1,C\rangle:V\longrightarrow\langle B_1',C'\rangle:V'}{\mathcal{P}\vdash\langle B_1\cup B_2,C\rangle:V\longrightarrow\langle B_1'\cup B_2,C'\rangle:V'}\qquad\text{(i)}$$

$$\frac{\mathcal{P}\vdash\langle\{\overline{b}=\overline{h_i}\}\cup G_i,C\rangle:(V\cup\mathcal{V}_{(h_i,G_i)})\xrightarrow{*}\langle\emptyset,C\cup C_g\rangle:V'}{\{h_i\,\mathtt{:-}\,G_i\mid B_i\}\cup\mathcal{P}\vdash\langle\{b\},C\rangle:V\longrightarrow\langle B_i,C\cup C_g\rangle:(V'\cup\mathcal{V}_{B_i})}\quad\begin{pmatrix}\text{if }\mathcal{E}\models\forall.(C\Rightarrow\delta\mathcal{V}_b.C_g)\\ \text{and }\mathcal{V}_{(h_i,G_i,B_i)}\cap V=\emptyset\end{pmatrix}\qquad\text{(ii)}$$

$$\mathcal{P}\vdash\langle\{t_1=t_2\},C\rangle:V\longrightarrow\langle\emptyset,C\cup\{t_1=t_2\}\rangle:V\qquad\text{(iii)}$$

規則 (i) は，ゴールのマルチ集合の並行書換えを表現している．ゴールのマルチ集合を書き換えるには，書換えのできる任意の部分集合 (もっと限定すれば，任意のゴール) を見つけて書き換えればよい．

規則 (ii) は，単一化 $\overline{b}=\overline{h_i}$ および G_i 中のガードゴールを，b 中の変数に影響を与えずに空集合になるまで書き換えることができるならば，非単一化ゴール b は，ガードつき

節 "h_i :- G_i | B_i" を用いて B_i に書き換えてよいことを表わす．付帯条件の前半が「b 中の変数に影響を与えずに」に対応するが，これは頭部の単一化がマッチングに制限されていることを意味する．付帯条件の後半 $\mathcal{V}_{(h_i,G_i,B_i)} \cap V = \emptyset$ は，ガードつき節中の変数が，新しい変数で置き換わっていることを保証する．頭部の単一化とガードゴールに用いるプログラム $\mathcal{P}$ は，それ自身，節 "h_i :- G_i | B_i" を含んでいてよいことに注意．

規則 (iii) は，単一化ゴールは単に，同形の制約を現在の束縛環境に追加することを述べている．

これらの規則から直ちに導くことのできる事実は，束縛環境の単調性である．つまり，もし $\langle B,C\rangle{:}V \longrightarrow \langle B',C'\rangle{:}V'$ ならば，$C' \supseteq C$ が $V' \supseteq V$ と共に成立する．

1.2.3 標準形

Flat GHC のプログラムを解析するには，プログラム節およびゴール節に出現する単一化ゴールの形を標準化し，下記の標準形条件を満たすようにしておくと好都合である:

(i) ガードには単一化ゴールが出現しない
(ii) 節のボディの単一化ゴールは，$v_1 = t_1, \ldots, v_n = t_n$ という形である．ただし

- 各 v_i は，節の頭部に現れる相異なる変数
- 各 v_i は，$t_1, \ldots, t_n$ にも，ボディの他のゴールにも出現しない
- もしある t_i が変数ならば，それが頭部にも出現する

ゴール節に対しては，条件 (i) は単に無関係で，条件 (ii) は，単一化ゴールをもたないことを意味する．条件 (i) があるので，次節以下で単に単一化ゴールと言ったときには，常に単一化ボディゴールを指すものとする．

我々はふつう，標準形条件を満たすようにプログラムを書いている．たとえば図 1.2 のプログラムも標準形である．標準形でないプログラムは，以下のようにして各節を標準形に変換する．プログラム節を

$$h \text{ :- } G_U \cup G_N \mid B_U \cup B_N$$

としよう．ここで，h は頭部，G_U は単一化ガードゴールのマルチ集合，G_N は非単一化ガードゴールのマルチ集合，B_U は単一化ボディゴールのマルチ集合，B_N は非単一化ボディゴールのマルチ集合である．もし $G_U \cup B_U$ が解をもたなければ，節は標準化できない．標準化できない節をもつプログラムのモード解析は考えないこととする．

解をもつ場合は，まず G_U を "実行" して，節を下記のように変形する:

$$h\sigma \text{ :- } G_N\sigma \mid B_U\sigma \cup B_N\sigma$$

ここで σ は，連立方程式 G_U のベキ等な mgu である．

次に，単一化ボディゴール $B_U\sigma$ を "実行" する．θ を，下記の条件を満たすような，$B_U\sigma$ のベキ等 mgu ($v \leftarrow t$ の形の結合の集合で表現) とする:

$$\forall (v \leftarrow t) \in \theta \, (v \in \mathcal{V}_{h\sigma} \wedge t \in Var \Rightarrow t \in \mathcal{V}_{h\sigma})$$

```
nt_node([],                  _, _, L,R) :- true | L=[], R=[].
nt_node([search(K,V)|Cs],K, V1,L,R) :- true |
        V=V1,                    nt_node(Cs,K,V1,L,R).
nt_node([search(K,V)|Cs],K1,V1,L,R) :- K<K1 |
        L=[search(K,V)|L1], nt_node(Cs,K1,V1,L1,R).
nt_node([search(K,V)|Cs],K1,V1,L,R) :- K>K1 |
        R=[search(K,V)|R1], nt_node(Cs,K1,V1,L,R1).
nt_node([update(K,V)|Cs],K, _, L,R) :- true |
                                 nt_node(Cs,K,V,L,R).
nt_node([update(K,V)|Cs],K1,V1,L,R) :- K<K1 |
        L=[update(K,V)|L1], nt_node(Cs,K1,V1,L1,R).
nt_node([update(K,V)|Cs],K1,V1,L,R) :- K>K1 |
        R=[update(K,V)|R1], nt_node(Cs,K1,V1,L,R1).

t_node([]                ) :- true | true.
t_node([search(K,V)|Cs]) :- true | V=undefined, t_node(Cs).
t_node([update(K,V)|Cs]) :- true |
        nt_node(Cs,K,V,L,R), t_node(L), t_node(R).
```

図 1.2: プロセスの 2 分木を定義したプログラム

言いかえると，θ 中のいかなる変数どうしの結合も，$h\sigma$ 中の変数を，$h\sigma$ の中にない変数に置き換えることはない．この性質を満たす mgu θ は，$B_U\sigma$ の任意のベキ等な mgu をもとに構成できる [43].

代入 η のアトム a への制限を $\eta|_a$ と書く．これは次のように定義される:

$$\eta|_a = \{\, (v \leftarrow t) \in \eta \mid v \in \mathcal{V}_a \,\}$$

また，η がベキ等で，$\bigcup_{i=1}^{n}\{v_i \leftarrow t_i\}$ の形をしているとき，$\overline{\eta}$ で連立方程式

$$\bigcup_{i=1}^{n}\{v_i = t_i\}$$

を表わす．これは明らかに，ベキ等な mgu η をもつ．

さて，$\overline{\theta|_{h\sigma}}$ を考えると，これはこの節の出力単一化を表わしていることがわかる．そこで，節の標準形は

$$h\sigma \text{ :- } G_N\sigma \mid \overline{\theta|_{h\sigma}} \cup B_N\sigma\theta$$

となる．

代入 θ はガードには適用しなくてよいことに注意してほしい．これは，単一化ボディゴールを実行するときにはガードの実行が終ってしまっているからである．

ゴール節

$$\text{:- } B_U \cup B_N$$

を標準化するには，単に B_U を "実行" すればよい．もし B_U が解をもたなければ，節は標準化できない．解をもつならば，節の標準形は

$$:- B_N\theta$$

となる．ここで θ は，B_U のベキ等な mgu である．

1.3 モード体系

Flat GHC のモード体系の目的は,「データ構造の各部分の値がもし具体化するならば，どのゴールが具体化するのか」を推論することである．これは，ゴール呼出し時や成功時の引数の具体化状態を推論するのが目的のモード解析 [11] とは，目的を異にする．このモード体系は，第 1.10 節で述べるように，PARLOG [8] や DEC-10 Prolog [3] のモードとも非常に異なる．Flat GHC のためのモード体系は [44] で最初に提案されたが，その表現や操作，正当性，解析の効率等はまだ研究途上であった．

我々のモード体系は静的解析を意図しているので，すべての無意味でない Flat GHC プログラムのデータの流れを解析することは不可能である．そこで我々は，プログラマが下記のプログラミング・スタイルを採っているものと仮定する:

(1) プロセス間通信は，競争的ではなく協調的である．つまり，プログラムの実行中に，同じ変数のいくつかの出現が作られたとき，そのうちのちょうど 1 個が，トップレベルの関数記号を決定できる**出力**出現で，残りはすべて**入力**出現である．言いかえると，ゴール節は，最初に与えられたものであれ，書換えの繰り返しによって得られたものであれ，各変数につきちょうど 1 個の出力出現をもつ．

(2) ゴール g の中の変数の出現のモードは，g の述語記号と，その出現を含むすべての項の主関数記号 (これらを祖先の記号と呼ぶ) のみに依存して決まる．このことは，述語の引数のモードは一意的に決まるものの，関数の引数のモードは，その関数がどこに出現するかに依存しうることを示している．例えば，図 1.2 のプログラムに現れるコマンド `search(K,V)` と `update(K,V)` を考えてみよう．第 2 引数 `V` のモードは，コマンド名に依存して決まってもよい (現にコマンド名によって異なる) が，第 1 引数 `K` の値に依存してはならない．例外は単一化のための組込述語 '=' で，これについては出現 (呼出し) ごとにモードが違っていてもよい．述語 '=' は，この意味でモードが**多義**である (overloaded) という．

Flat GHC へのモード体系の導入は，実質的に Flat GHC のサブセットの設定になっている．このサブセットを *Moded Flat GHC* という．問題は，この仕様縮小が GHC プログラミングにとって深刻な影響を及ぼすかどうかであるが，これまで書かれたほとんどの GHC プログラムは，上記のスタイルで書かれているか，上記のスタイルに容易に書き換えることができる．その一つの理由は，GHC ではボディの単一化が失敗すると回復手段がないことによる．

仮定 (1) に関して，既存のプログラムの中には，"アボート信号" 技法を使ったものがあることにふれておく必要がある．この技法は，いくつかのプロセス $p_1,\ldots,p_n$ が変数 v を共有するとともに，v と結合する定数 c について合意しておき，ある p_i が，他のプロ

セスが計算をやめていいと判断した時点で v を c に具体化する，というものである．すべての p_i が v を具体化する可能性があるが，単一化の失敗は起きえない．このようなプログラムを仮定 (1) に合ったスタイルにする一つの方法は，n 引数のアービタプロセスを起動しておき，個々の p_i は，(相異なる変数を通じて) v を c に具体化するように，アービタに依頼するというものである．このようなアービタは，メッセージ指向処理方式をとれば効率良く実現できる [46].

仮定 (2) に反するプログラムはあまりないが，図1.2のプログラムの中の項 `search(K,V)` と `update(K,V)` をそれぞれ `[search,K,V]` と `[update,K,V]` に書き換えることによって人為的に作ることができる．この変更を施すと，`search` と `update` が `K` や `V` の祖先の記号ではなくなってしまい，二つのコマンドの引数のモードが区別できなくなる．つまり仮定 (2) は，プログラマに意味のある関数記号だけを使うようにさせ，リストの濫用を慎ませる効果がある．これらの仮定は，静的モード解析を可能にするだけでなく，良いGHCプログラムの指針としても役立つであろう．

さて，プログラムの記述や実行に使う述語，関数，変数記号の (共通部分を持たない) 集合をそれぞれ $Pred$, Fun, Var としよう．また $Pred$, Fun, Var から作られるアトムと項の集合をそれぞれ $Atom$, $Term$ と書く．

n_p 引数の各 $p \in Pred$ に対し，N_p で集合 $\{1, 2, \ldots, n_p\}$ を表わす．各 $f \in Fun$ に対して同様に N_f を定義する．さらに，パスの集合 P_{Term} (項用) と P_{Atom} (アトム用) を，依存型 (dependent type) を用いて次のように定義する:

$$P_{Term} \stackrel{\text{def}}{=} (\sum_{f \in Fun} N_f)^*, \quad P_{Atom} \stackrel{\text{def}}{=} (\sum_{p \in Pred} N_p) \times P_{Term}$$

P_{Term} の要素は列 $\langle f_1, j_1 \rangle \ldots \langle f_n, j_n \rangle$ で，また P_{Atom} の要素は $\langle p, i \rangle q$ ($q \in P_{Term}$) と表現できる．P_{Term} の要素である空列は，ϵ と書く．パスの目的は，項やアトムやそれらの具体形 (instance) を構成する変数記号や関数記号を指定することである．そこで，個々の項 t に対し，その構成記号にパスを用いてアクセスするための関数 $\tilde{t} : P_{Term} \to Var \cup Fun \cup \{\bot\}$ ($\bot$ は "未定義" を表わす) を下記のように定義する:

$$\begin{cases} \tilde{t}(\epsilon) \stackrel{\text{def}}{=} \begin{cases} f, & t \text{ が } f(t_1, \ldots, t_n) \text{ の形のとき} \\ t, & \text{それ以外 (つまり } t \text{ が変数のとき)} \end{cases} \\ \tilde{t}(\langle f, j \rangle q) \stackrel{\text{def}}{=} \begin{cases} \tilde{t_j}(q), & t \text{ が } f(t_1, \ldots, t_n) \text{ の形で } j \in N_f \text{ のとき} \\ \bot, & \text{それ以外のとき} \end{cases} \end{cases}$$

アトムの構成記号にアクセスする関数も同様に定義する．

我々の枠組では，パスの各要素を依存型で表現することによって，述語記号や関数記号を陽に示している．これは，項やアトムのすべての可能な具体形の構成記号にアクセスできるようにするためである．

最後にモードを定義する．**モード**の集合 M は，

$$M \stackrel{\text{def}}{=} P_{Atom} \to \{in, out\}$$

と定義される．ここで $in \neq out$ と仮定する．この定義は，すべてのゴールのすべての可能な具体形のすべてのパスに対して，in か out のいずれかを割り当てるものである．

記述の簡潔化のために，以下のような記法も定めておこう:

- モード $m \in M$ とパス $p \in P_{Atom}$ に対し，m/p は，次のような性質を満たす P_{Term} から $\{in, out\}$ への関数を表わす:

$$\forall q \in P_{Term}((m/p)(q) = m(pq))$$

これを m の**部分モード**と呼ぶ．部分モードの部分モードも同様に定義する．

- モード m の**逆モード** $\overline{m}$ とは，

$$\forall p \in P_{Atom}(\overline{m}(p) \neq m(p))$$

を満たす関数である．部分モードの逆モードも同様に定義する．

- IN を

$$\forall q \in P_{Term}(IN(q) = in)$$

を満たす部分モードとし，OUT を $\overline{IN}$ と定義する．

1.4 モード解析

モード解析の目的は，プログラム $\mathcal{P}$ とそれを起動するゴール節 G との対が構文的に課するすべての制約 (後で列挙) を満たすモード $m \in M$ を求めることである．もしそのようなモードがあるならば，$\mathcal{P}$ と G の対は **well-moded** であるといい，m をこの対の **well-moding** という．Well-moded という概念は，$\mathcal{P}$ と G のそれぞれや，さらに一般に (プログラムまたはゴール) 節の集合—**プログラムの断片**という—に対しても同様に定義する．

プログラムの断片は，well-moded でないこともある．これは，あるパスのモード値が in と out の両方に制約されたとき，あるいはあるパスのモード値がそのモード値自身と逆の値に制約されたときなどに起きる．

また，well-moded なプログラムの断片は，通常多数の well-moding をもつ．これは，値を検査したり具体化したりすることのないパスには，どちらのモード値を与えてもかまわないからである．そこで我々は，モードを具体的な関数形でなく，モード制約の集合の形で表現する．

制約に基づく表現を使うと，それを簡約化したり，制約集合が解をもつか否かを判断したりするための制約充足アルゴリズムが必要となる．しかし，制約充足アルゴリズムの設計はモード体系とは別の問題であり，第 1.6 節および第 1.7 節で述べることとする．

モード解析は，標準形のプログラム断片に対して行なう．厳密に言えば，第 1.2.3 節の標準形条件 (ii) はモード解析に必須のものではない．しかしこれによって，well-moded ないプログラムを，意味を変えずに well-moded なプログラムに変換できることがある．プログラムの実行による書換えの結果得られたゴール節は，条件 (ii) を満たさないかもしれないことに注意してほしい．これは，非単一化ゴールの書換えによって単一化ゴールが生成されるかもしれないからである．

プログラムを標準化しておく以外の準備として，述語 '=' のモードの多義性に対処するため，そのすべての出現が '$=_1$', '$=_2$', ... と添字づけられているものとみなす．この扱い

が必要なのは，Flat GHCでは，異なる単一化ボディゴールが異なるモードを持つのがごく普通だからである．幸いにして，この扱いによって解析が複雑になることはない．述語'='の意味が決まっていて，その引数のモードには強い制約がかかるからである(下記の制約(BU)参照)．なお，ゴール節の書換え時に生成した単一化ゴールは，プログラム中の対応する単一化述語の添字を継承するものとする．

さて以下に，プログラムの断片中の各節Cが課する制約を列挙する[1]．Cは，"h :- G | B" (プログラム節)または ":- B" (ゴール節)という形をしている．非テスト述語を定義する節には，以下のすべての制約を適用するが，テスト述語に対しては，(HF)(HV)(GV)のみを適用する[2]．ゴール節はガードを持たないので，(BU)(BF)(BV)のみが適用される．

(HF) $\forall p \in P_{Atom}(\tilde{h}(p) \in Fun \Rightarrow m(p) = in)$
(h 中の p にある記号が関数記号ならば $m(p) = in$)

(HV) $\forall p, p' \in P_{Atom}(\tilde{h}(p) \in Var \wedge \tilde{h}(p) = \tilde{h}(p') \Rightarrow m/p = IN)$
(h 中の p にある記号が，h 中に2回以上出現する変数ならば
$\forall q \in P_{Term}(m(pq) = in)$，つまり $m/p = IN$)

(GV) $\forall p, p' \in P_{Atom}\, \forall a \in G(\tilde{h}(p) \in Var \wedge \tilde{h}(p) = \tilde{a}(p')$
$\Rightarrow \forall q \in P_{Term}(m(p'q) = in \Rightarrow m(pq) = in))$
(同じ変数が h 中の p と G 中の p' に出現するならば
$\forall q \in P_{Term}(m(p'q) = in \Rightarrow m(pq) = in)$)

(BU) $\forall k > 0\, \forall t_1, t_2 \in Term((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle})$
(単一化ボディゴールの二つの引数は，全く逆の部分モードを持つ)

(BF) $\forall p \in P_{Atom} \forall a \in B(\tilde{a}(p) \in Fun \Rightarrow m(p) = in)$
(ボディゴール中の p にある記号が関数記号ならば $m(p) = in$)

(BV) 変数 v が，h と B の中にちょうど $n\,(\geq 1)$ 回，$p_1, \ldots, p_n$ に出現し，そのうち h 中の出現が $p_1, \ldots, p_k$ $(k \geq 0)$ であるとする．このとき

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & k = 0 \text{ のとき} \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & k > 0 \text{ のとき} \end{cases}$$

ただし述語 $\mathcal{R}$ は"協調関係"を表わしたもので，部分モードの**マルチ集合**に対して次のように定義される:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S(s(q) = out \wedge \forall s' \in S \backslash \{s\}\, (s'(q) = in)) \qquad \blacksquare$$

制約(BV)は，v の h 内での2回目からの出現と，G 中のすべての出現を無視することに注意．それ以外の v の出現を，**チャネル出現**と呼ぶ．また，上の $\mathcal{R}$ の定義で，s が q

[1]制約の名称 (HF)(HV)(GV)(BU)(BF)(BV) は，*head function, head variable, guard variable, body unification, body function, body variable* をそれぞれ表わす．

[2]制約 (BU) と (BF) はテスト述語には適用不可能であるから，ここでの要点は (BV) を適用しないということである．

に依存して決まることにも注意．制約 (BV) は直観的には，v がとりうる具体形の中に出現するどの各関数記号も，v の出現のうちのどれか一つが決定することを述べている．

関係 $\mathcal{R}$ は下記のような性質を満たす:

$$\mathcal{R}(\{s\}) \Leftrightarrow s = OUT \tag{1.1}$$
$$\mathcal{R}(\{s_1, s_2\}) \Leftrightarrow s_1 = \overline{s_2} \tag{1.2}$$
$$\mathcal{R}(\{IN\} \cup S) \Leftrightarrow \mathcal{R}(S) \tag{1.3}$$
$$\mathcal{R}(\{OUT\} \cup S) \Leftrightarrow \forall s' \in S(s' = IN) \tag{1.4}$$
$$\mathcal{R}(\{s, s\} \cup S) \Leftrightarrow s = IN \wedge \mathcal{R}(S) \tag{1.5}$$
$$\mathcal{R}(\{\overline{s}, s\} \cup S) \Leftrightarrow \forall s' \in S(s' = IN) \tag{1.6}$$
$$\mathcal{R}(\{\overline{s}\} \cup S_1) \wedge \mathcal{R}(\{s\} \cup S_2) \Rightarrow \mathcal{R}(S_1 \cup S_2) \tag{1.7}$$
$$\mathcal{R}(\bigcup_{1 \le i \le n}\{s_i\}) \Rightarrow \mathcal{R}(\bigcup_{1 \le i \le n}\{s_i/q\}), \quad q \in P_{Term} \tag{1.8}$$

証明は容易である．性質 (1.7) は，Robinson の融合原理と相似のものである．

性質 (1.1) と (1.2) は，v のチャネル出現が高々二つならば，制約 (BV) がずっと簡単になることを示している．v がちょうど二回，p_1 と p_2 にチャネル出現する—それが最も普通である—ときは，(BV) は，それらの一方が頭部に現れるか，両方ともボディに現れるかによって，$m/p_1 = m/p_2$ または $m/p_1 = \overline{m/p_2}$ と等価となる．これは，v が 1 対 1 通信に使われていることを意味する．変数 v が 1 回だけ，p にチャネル出現する場合は，(BV) は，その出現が頭部の中かボディ中かによって，$m/p = IN$ または $m/p = OUT$ と等価となる．

1.5 制約規則の根拠

前節に挙げた 6 個のモード制約の根拠について述べる．まず非テスト述語から考えることとし，テスト述語は全く異なる性質を持つので後述することとする．

並行論理プログラミングでは，ボディゴール (より正確には，ボディゴールによって実現されるプロセス) は，非テスト述語によって定義され，**端子**と呼ばれる情報の出入口をもった情報処理装置とみなすことができる．変数は，その出現どうしを結ぶ 1 対 n ($n \ge 0$) の通信チャネルであり，その個々の出現は，ゴールの端子に接続されていると考えることができる．プログラム節の頭部とボディの双方に出現する変数は，その頭部がマッチするゴールと，サブゴールとの間で情報を運ぶチャネルと考えればよい．関数記号は，非可分 (atomic) 情報の湧出口 (outlet，ボディに出現の場合) または吸込口 (inlet，頭部に出現の場合) として機能する，接続線のないプラグと考えればよい．電気機器のチャネルや端子は，ふつう配列構造をもつが，我々のチャネルや端子は入れ子構造をもつ．つまり，$p_1, \ldots, p_n$ にある端子を結ぶ変数は，$p_1 q, \ldots, p_n q$ ($q \in P_{Term}$) どうしも結ぶ．

ゴールの端子には必ず**相手**がある．非単一化ゴールの呼出し側の p にある端子の相手は，呼び出された側 (節の頭部) の同じ場所にある端子である．また単一化ゴールの第 1 引数の $\langle =_k, 1\rangle q$ にある端子の相手は，第 2 引数の $\langle =_k, 2\rangle q$ にある端子である．ゴールの書換えは，相互の接続が確立した端子対の消去と考えることができる．

モード制約は，(i) チャネル内，および (ii) 端子 における情報の流れの方向に注目して課したものである．各規則の背後にある基本原理は，次の二つである:

(i) チャネル (つまり変数) が n 個の端子 (うち節の頭部にあるのは高々一つ) を接続しているとき，そのうちのちょうど一つが情報の湧出口であり，他は吸込口である．つまり，節の中に n 回チャネル出現する変数は，1 対 $(n-1)$ 通信に使われる．
(ii) ゴールの相手どうしの端子のうち，ちょうど一つが情報の湧出口であり他方は吸込口である．

まず制約 (BV) は，原理 (i) から来ている．変数の節頭部における入力 (出力) 出現は，節の内部から見れば情報の湧出 (吸込) 口と考えることができる．このため，制約 (BV) を考えるときに，頭部に関するモードを逆転させるのである．制約 (BV) は，変数 v の頭部の出現のうちの一つしか考慮しない．同じ変数が頭部に複数回現れるのは，コミット**前**に同一性検査を行なう必要のあるときであり，コミット**後**は，その変数がボディにも出現してボディゴールに情報を運ぶか否かだけが問題となる．

制約 (HF) および (HV) は原理 (ii) から来ている．非単一化ゴールについては，ある節でパス p の値を検査するかも知れないときは $m(p)$ の値を in に制約しなければならない．なぜならこのような検査は，ゴールの**呼び出された**側の，情報の湧出口で行わなければならないからである．(HV) の強い制約は，節の頭部に変数が複数回出現するときは，それらが (何でもよいが) 同一の項を呼出し側から受け取らなければならない，という Flat GHC の意味論から来ている．

制約 (BU) は原理 (ii) を単一化ボディゴールに適用したものにほかならない．一方の引数のあるパス $\langle =_k, i\rangle q$ から供給した値は，他方の引数の対応するパス $\langle =_k, 3-i\rangle q$ から出てくる．

制約 (BF) も原理 (ii) から来ている．ゴールの呼出し側の関数記号は，情報の吸込口にしか出現できない．なぜならその値についての情報が対応する湧出口から出てゆくからである．

さて，テスト述語が定義するガードゴールについての制約 (GV) を考えてみよう．基本的な考え方は，ゴール g が呼び出した節のガードゴールが検査するかも知れないパスは，in に制約する，というものである．制約 (GV) の後件部に対する代案は，$\forall q{\in}P_{Term}(m(p'q) = in \Rightarrow m(pq) = in)$ のかわりに，もっと強い形 $m/p' = m/p$ を使うというものであろう．しかし，テスト述語はモードに関して汎用であるべきで，我々の制約規則は，非テスト述語からテスト述語へは制約が伝播しないように与えている．

ほとんどすべての Flat GHC の処理系は，ガードゴールを**組込みの**テスト述語の呼出しに制限している．これらの述語は，仮想的に

```
6>5 :- true | true
```

のような節の集合で定義してあるとみなし，それらの節に対してモード制約を考えればよい．

ガードゴールがあるときは，制約 (BV) を弱くすることができる．変数 v が，節 C の頭部とガードゴールの双方に出現するとしよう．するとそれらのガードゴールは,「あるゴールが節 C にコミットしたときには，v がある非変数項に具体化している」ことを保

証してくれるかも知れない．例えば，Cが

```
p(X, ...) :- X>0 | ...
```

という形のときは，Xはコミット時には定数に具体化しているはずである．このような場合，パス$\langle \mathrm{p}, 1\rangle q$ ($q \in P_{Term} \backslash \{\epsilon\}$) に対する制約は余分であり，$\mathcal{R}$の定義に現れる束縛変数の値は，これらのパスの上を動く必要はない．

一般化しよう．項の集合T_vを，節Cへのコミット時にvがとりうる値を含む集合とし，P_{T_v}で集合

$$P_{Term} \backslash \{ p \in P_{Term} \mid \forall t \in T_v (\tilde{t}(p) = \bot) \}$$

を表わす．直観的には，P_{T_v}は無関係なパス (の一部) を除外するものである．このP_{T_v}を用いると，述語$\mathcal{R}$の定義を次のように少し変更できる:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{T_v}\, \exists s \in S (s(q) = out \wedge \forall s' \in S \backslash \{s\}\, (s'(q) = in))$$

もし$T_v = Term$ならば，つまりvのとりうる値に関して何も知識がなければ，P_{T_v}はP_{Term}と同じであることに注意してほしい．一方，X>0の例のように，T_vとして定数の集合が使えることが推論できるならば，P_{T_v}は要素1個の集合$\{\epsilon\}$となる．小さなT_vを見つけることにより，モード解析系が，不必要なモード制約を課して意味のあるプログラムを排除するのを防ぐことができる．しかし，第1.8節に述べるような，well-modedなプログラムが満たす諸性質は，プログラム解析系が正しいT_vを使う限り，どのようなT_vを使うかには無関係に成立する．

1.6 制約の管理と操作

本節では，単純なプログラムのモード制約が素性グラフで表現でき，その単一化によって操作できることを，例を用いて示す．この例がカバーしない一般の場合については，次節で考える．

図1.3のプログラムは，単純なスタックプロセスとそのドライバの定義である[3]．述語terminateは，スタックプロセスの終了時に，スタック内に残った要素を処理するためのものである．このプログラムでは，スタックを基底項 (整数) の格納に使っているだけなので，terminateは単にすべての要素を捨てるだけでよい．しかし，スタックが，他のプロセスにつながっているストリームを格納しているような場合は，スタックプロセスの終了時に残ったストリームを閉じて，受信プロセスが終了できるようにしなければならない．

このプログラムは，単一化以外に三つの組込述語'=:=', '=\=', subtractを使っているので，それらが課するモード制約を知る必要がある．これらが整数式ではなく，整数を引数にとると仮定すると，モード制約は下記の通りである:

$$m(\langle \texttt{=:=}, i\rangle) = m(\langle \texttt{=\=}, i\rangle) = in, \quad i = 1, 2$$

$$m(\langle \texttt{subtract}, 1\rangle) = m(\langle \texttt{subtract}, 2\rangle) = in, \quad m(\langle \texttt{subtract}, 3\rangle) = out$$

[3] stackの第2引数 (スタック要素記憶用) はリスト構成子を使っていない．これは，ストリームに対して特別の最適化を施す処理系においては，ストリームの構成子と非ストリームデータの構成子とを区別する必要があることを想定したものである．しかしモード解析に話を限定すれば，第2引数にリスト構成子を使っても，解析にはまったく支障がない．

```
drive(M,S) :- M=:=0 | S=1[].                                          (i)
drive(M,S) :- M=\=0 |
    S=2[push(M),pop(N)|S1], subtract(N,1,N1), drive(N1,S1).           (ii)

stack([],          D       ) :- true | terminate(D).                  (iii)
stack([push(X)|S],D        ) :- true | stack(S,p(X,D)).               (iv)
stack([pop(X)|S], p(Y,D1)) :- true | X=3Y, stack(S,D1).               (v)

terminate(D) :- true | true.                                          (vi)
```

図 1.3: スタックとそのドライバ

空でないリストの関数記号を '.' で表わすことにすると，述語 drive から直接得られるモード制約は次の通りである:

(1) $m(\langle =:=,1\rangle) = in \Rightarrow m(\langle \mathtt{drive},1\rangle) = in$ ((GV) を節 (i) の M に適用)

(2) $m/\langle =_1,1\rangle = \overline{m/\langle =_1,2\rangle}$ ((BU) を節 (i) の '$=_1$' に適用)

(3) $m(\langle =_1,2\rangle) = in$ ((BF) を節 (i) の [] に適用)

(4) $m/\langle \mathtt{drive},1\rangle = IN$ ((BV) を節 (i) の M に適用)

(5) $m/\langle \mathtt{drive},2\rangle = m/\langle =_1,1\rangle$ ((BV) を節 (i) の S に適用)

(6) $m(\langle =\backslash =,1\rangle) = in \Rightarrow m(\langle \mathtt{drive},1\rangle) = in$ ((GV) を節 (ii) の M に適用)

(7) $m/\langle =_2,1\rangle = \overline{m/\langle =_2,2\rangle}$ ((BU) を節 (ii) の '$=_2$' に適用)

(8) $m(\langle =_2,2\rangle) = in$ ((BF) を節 (ii) の外側の '.' に適用)

(9) $m(\langle =_2,2\rangle\langle .,1\rangle) = in$ ((BF) を節 (ii) の push に適用)

(10) $m(\langle =_2,2\rangle\langle .,2\rangle) = in$ ((BF) を節 (ii) の内側の '.' に適用)

(11) $m(\langle =_2,2\rangle\langle .,2\rangle\langle .,1\rangle) = in$ ((BF) を節 (ii) の pop に適用)

(12) $m(\langle \mathtt{subtract},2\rangle) = in$ ((BF) を節 (ii) の 1 に適用)

(13) $m/\langle \mathtt{drive},1\rangle = m/\langle =_2,2\rangle\langle .,1\rangle\langle \mathtt{push},1\rangle$ ((BV) を節 (ii) の M に適用)

(14) $m/\langle \mathtt{drive},2\rangle = m/\langle =_2,1\rangle$ ((BV) を節 (ii) の S に適用)

(15) $m/\langle =_2,2\rangle\langle .,2\rangle\langle .,1\rangle\langle \mathtt{pop},1\rangle = \overline{m/\langle \mathtt{subtract},1\rangle}$ ((BV) を節 (ii) の N に適用)

(16) $m/\langle \mathtt{subtract},3\rangle = \overline{m/\langle \mathtt{drive},1\rangle}$ ((BV) を節 (ii) の N1 に適用)

(17) $m/\langle =_2,2\rangle\langle .,2\rangle\langle .,2\rangle = \overline{m/\langle \mathtt{drive},2\rangle}$ ((BV) を節 (ii) の S1 に適用)

述語 '=' や他の組込述語の制約を消去すると，$m/\langle \mathtt{drive},1\rangle$ と $m/\langle \mathtt{drive},2\rangle$ に対する下記の制約が得られる:

$$\begin{aligned} m/\langle \mathtt{drive},1\rangle &= IN \\ m(\langle \mathtt{drive},2\rangle) &= out \\ m(\langle \mathtt{drive},2\rangle\langle .,1\rangle) &= out \end{aligned}$$

$$
\begin{aligned}
m/\langle \mathtt{drive},2\rangle\langle .,1\rangle\langle \mathtt{push},1\rangle &= \overline{m/\langle \mathtt{drive},1\rangle} \\
m(\langle \mathtt{drive},2\rangle\langle .,2\rangle) &= out \\
m(\langle \mathtt{drive},2\rangle\langle .,2\rangle\langle .,1\rangle) &= out \\
m(\langle \mathtt{drive},2\rangle\langle .,2\rangle\langle .,1\rangle\langle \mathtt{pop},1\rangle) &= in \\
m/\langle \mathtt{drive},2\rangle\langle .,2\rangle\langle .,2\rangle &= m/\langle \mathtt{drive},2\rangle
\end{aligned}
$$

例えば，$m(\langle \mathtt{drive},2\rangle) = out$ は (2)，(3)，(5) から導かれ，$m/\langle \mathtt{drive},2\rangle\langle .,2\rangle\langle .,2\rangle = m/\langle \mathtt{drive},2\rangle$ は (7)，(14)，(17) から導かれる．

もし，M の値が定数以外ならば M=:=0 と M=\=0 が成功しないことを解析系が知っていたら，(4) の $m/\langle \mathtt{drive},1\rangle = IN$ のかわりに，制約 (BV) の弱い版 (第 1.5 節) から $m(\langle \mathtt{drive},1\rangle) = in$ を導くことができる．これは，(GV) から導ける制約と全く同一である．

同様に stack からは，下記の制約が得られる:

$$
\begin{aligned}
m(\langle \mathtt{stack},1\rangle) &= in \\
m(\langle \mathtt{stack},1\rangle\langle .,1\rangle) &= in \\
m/\langle \mathtt{stack},1\rangle\langle .,1\rangle\langle \mathtt{push},1\rangle &= m/\langle \mathtt{stack},2\rangle\langle \mathtt{p},1\rangle \\
m/\langle \mathtt{stack},1\rangle\langle .,1\rangle\langle \mathtt{pop},1\rangle &= \overline{m/\langle \mathtt{stack},2\rangle\langle \mathtt{p},1\rangle} \\
m/\langle \mathtt{stack},1\rangle\langle .,2\rangle &= m/\langle \mathtt{stack},1\rangle \\
m(\langle \mathtt{stack},2\rangle) &= in \\
m/\langle \mathtt{stack},2\rangle\langle \mathtt{p},2\rangle &= m/\langle \mathtt{stack},2\rangle \\
m/\langle \mathtt{terminate},1\rangle &= m/\langle \mathtt{stack},2\rangle
\end{aligned}
$$

これらすべての制約の連言 (conjunction) は，図 1.4 のように，(有向で，ループがあるかも知れず，非連結かもしれない) 素性グラフ [18] として表現することができるような素性構造を形成する．これを**モードグラフ**という．

我々の枠組では，素性構造の個々の素性は，述語または関数記号とその引数位置との対であり，図 1.4 では辺のラベルとして示してある．素性の列は，我々のモード体系の意味においても，グラフ理論的な意味においても，パスを形づくる．グラフの節点は，その節点で終わるパスのモード値をラベルとしてもつことがある．

黒玉のついた辺は，“モード反転” 辺である．パスの中に奇数個のモード反転辺があるとき，そのパスは**反転**しているといい，そのパスの終端についているモード値を反転させて解釈する．このように，パス中のモード反転辺の数は，パスの**極性**を決める．$m/p_1 = m/p_2$ ないしは $m/p_1 \neq m/p_2$ の形をした全称限量制約は，二つ (以上) の入力辺をもつ共有節点で表現できる．二つの入力辺の極性が異なるときは，共有節点は不等式に対応し，同じときは等式に対応する．

$m/p = IN$ ないしは $m/p = OUT$ の形の全称限量制約は，“基底” というラベルのついた節点で表現する．“基底” ラベルのついた節点は，その節点で終わるすべてのパスや，その節点を通って先に行こうとするすべてのパスのモード値を，in ないしは out に制約する．in と out のどちらになるかは，この節点に至るまでに通過したモード反転辺の数が偶数か奇数かによってきまる．

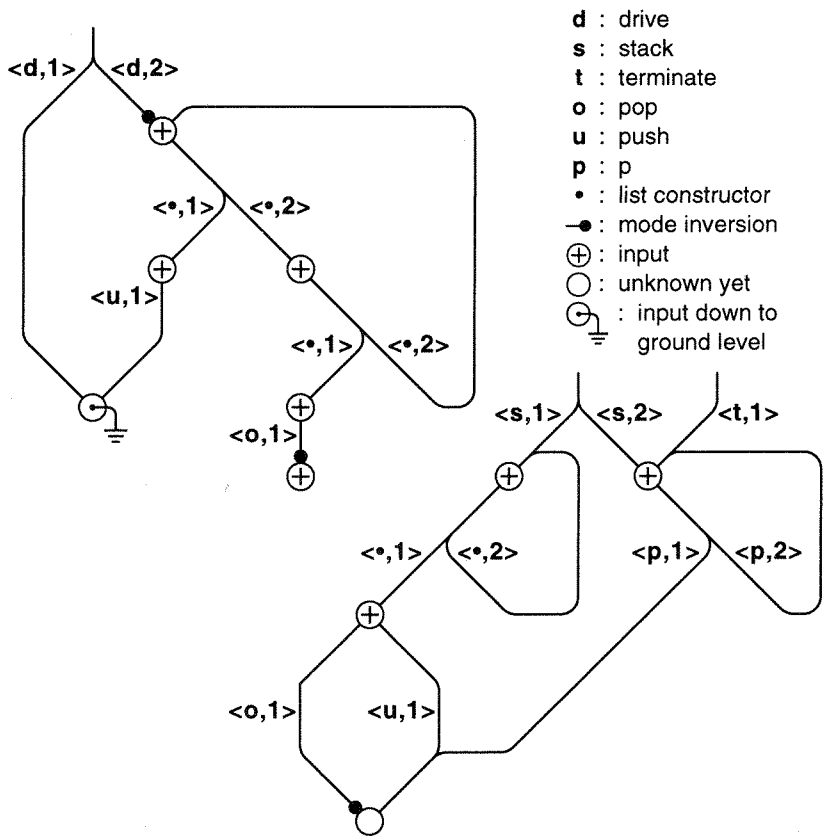

図 1.4: `drive` と `stack` から独立に得られるモード制約

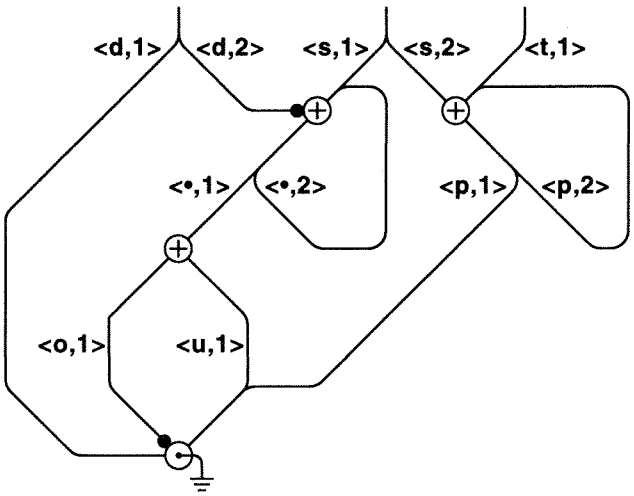

図 1.5: 新たな制約を併合した後のモード制約

モード m と，図1.4のようなモードグラフは，アトム全体のトップレベルのモード値に関する情報を省いているとみなすことができる．アトムの述語記号は常に呼出し側が決めるので，アトムのトップレベルのモード値は常に in と考えることができるからである．

モード m のグラフとパス p が与えられたとき，p にある節点と，そこから到達可能なすべての節点や辺は，部分モード m/p を表わす部分グラフを形づくる．これを m/p の**部分モードグラフ**と呼び，p にある節点をこの部分モードグラフの**根**という．

図1.4の `drive` のモードが示すように，我々の枠組では，ストリームの異なる要素が必ずしも同じモード制約をもつ必要はない．

$m(\langle \mathtt{stack},1\rangle\langle .,1\rangle\langle \mathtt{push},1\rangle)$，$m(\langle \mathtt{stack},1\rangle\langle .,1\rangle\langle \mathtt{pop},1\rangle)$，および $m(\langle \mathtt{stack},2\rangle\langle \mathtt{p},1\rangle)$ の具体値は，`stack` だけから決定することはできない．これらは述語 `stack` が使われる文脈や，`terminate` の定義を与えることによって初めて決まる．たとえば，ゴール節ないしは他のプログラム節が，ボディゴール `drive(10,S)` と `stack(S,none)` を持ち，`S` がその節の他の場所には出現しないとしよう．このとき，二つの部分モード $m/\langle \mathtt{drive},2\rangle$ と $m/\langle \mathtt{stack},1\rangle$ は完全に逆のモードに制約される．そこで $m(\langle \mathtt{stack},1\rangle\langle .,1\rangle\langle \mathtt{push},1\rangle)$ と $m(\langle \mathtt{stack},2\rangle\langle \mathtt{p},1\rangle)$ は in となり，$m(\langle \mathtt{stack},1\rangle\langle .,1\rangle\langle \mathtt{pop},1\rangle)$ は out となる．これらの制約を付加したモードグラフを図1.5に示す．

述語 `terminate` は，制約 (BV) から

$$m/\langle \mathtt{terminate},1\rangle = IN$$

という強いモード制約を課する．この制約を図1.5のモードグラフに併合すると，図1.6のモードグラフとなる．

述語のモードは，文脈を完全に与えても一意に決まるとは限らない．しかし，我々が必要とするのは，コード生成に関係してくる情報だけである．たとえば $m(\langle \mathtt{drive},2\rangle\langle f, i\rangle q)$ の値は，'.' 以外の関数記号 f，$i \in N_f$，$q \in P_{Term}$ のいかなる組に対しても決まらないが，それで困ることはない．

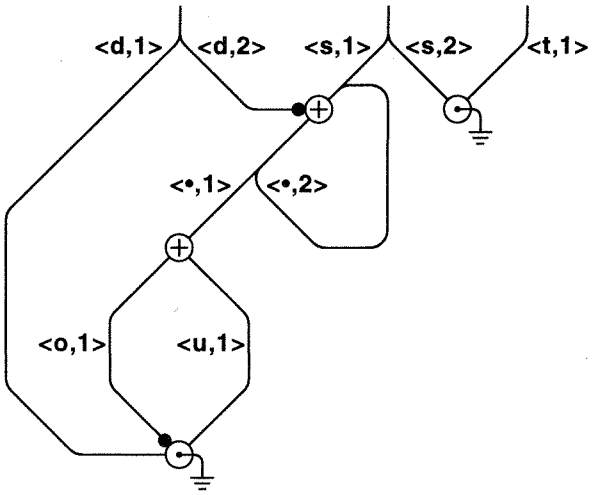

図 1.6: もう一つ制約を併合した後のモード制約

相互再帰で定義した述語は，これまで述べた方法でそのまま扱うことができ，別の手法を追加する必要はない．パスに基づくモード体系は，ストリームのストリームも扱うことができる．次の節を考えてみよう．

```
create_stacks([S|Ss]) :- true |
        stack(S,none), create_stacks(Ss)
```

この節からは，制約

$$m/\langle \texttt{create\_stacks}, 1\rangle\langle ., 1\rangle = m/\langle \texttt{stack}, 1\rangle$$
$$m/\langle \texttt{create\_stacks}, 1\rangle\langle ., 2\rangle = m/\langle \texttt{create\_stacks}, 1\rangle$$

が得られる．したがって，create_stacks の引数として出現するストリームの各要素は，stack の第 1 引数として出現するストリームの部分モードと共通の部分モードをもつ．

Well-moded なプログラムをゴール節で起動するときは，ゴール節が課するモード制約が，プログラムのモードと矛盾しないかどうかをまず検査しなければならない．

1.7 解析の計算量

本節では，モード解析の計算量について考察する．まずは単純だが重要な場合から考えよう．我々は当面，非テスト述語に焦点を当て，また次のことを仮定する:

- どの節もガードゴールをもたない
- どの節も，3 個以上のチャネル出現をもつ変数をもたない

これらの仮定は，モード制約が，下記の形の**原始モード制約**の集合として表現できることを保証する:

$$
\begin{array}{ll}
m(p) = in & (\text{または } m(p) = out) \\
m/p_1 = m/p_2 & (\text{または} m/p_1 \neq m/p_2) \\
m/p = IN & (\text{または } m/p = OUT)
\end{array}
$$

最初の形式は，モードグラフの中においては，pにある節点をinとラベルづけることで表現できる．二つめは，節点の共有によって表現できる．三つめは，pにある節点を“基底”とラベルづけることで表現できる．括弧内の形式は，極性の反転したパスを用いて表現できる．したがって，いま考えているプログラムのクラスに対しては，第1.6節で説明したモードグラフの道具立てだけで対処できる．個々の原始モード制約は，それ自身，非常に単純なモードグラフで表現できることにも注意してほしい．図1.4 ～ 1.6は，原始モード制約を表わした単純なモードグラフを多数併合した結果であると考えることができる．

1.7.1 二つのモードグラフの併合

図1.4 ～ 1.6が示唆するように，モードグラフとして表現した二つのモード制約集合の併合は，素性グラフの単一化[18]と非常に似ており，したがって有理項の単一化[15]とも非常に近い[4]．

ただし違いもある．それは，グラフ構造を単一化するだけでなく，対応するパスのモード値どうしを，パスの極性と節点のラベルの両方を考慮しつつ単一化しなければならないということである．

とりわけ，“基底”ラベルのついた節点の扱いは正確を期する必要がある．まず，“基底”ラベルをもつ節点一つからなる部分モードグラフどうしの単一化は，それらの“基底”節点に至るパスの極性が同じかどうかによって，成功または失敗する．次に，“基底”節点1個からなる部分モードグラフ$\mathcal{G}_1$と，根が“基底”節点でない部分モードグラフ$\mathcal{G}_2$とは，$\mathcal{G}_2$のすべての節点を“基底”とラベルづける(またはつけ直す)ことによって単一化できる．このとき，新たなラベルが，すでについているラベル(もしあれば)と矛盾しないことは，もちろん確かめる必要がある．新たに“基底”とラベルづけた節点からの出力辺は，削除してもよい．それによってすべての入力辺を失った節点はゴミとなり，そこからの出力辺も同じように削除してよい．ただし，$\mathcal{G}_2$の節点の中には，他のパスから到達可能なものがあるかも知れない．そのような節点はゴミとせずに，正しく(再)ラベルづけしなければならない．

時間計算量を検討するにあたって，まず，二つの素性グラフの単一化を，グラフの大きさlに関してほとんど線形の手間で行なう実用的なアルゴリズムがあることを指摘しておく[1, 15]．ほとんど線形とは，αをアッカーマン関数の逆関数として，$O(l \cdot \alpha(l))$であるということである．この単一化アルゴリズムは，union-findアルゴリズムで管理する不可視ポインタ(図1.4 ～ 1.6では陽に示していない)を用いて，節点の共有を表現する．共有節点へアクセスするとき，これらの不可視ポインタをたぐるかもしれないので，アルゴリズムがほとんど線形ではあるものの，完全に線形にはならないのである．

したがって，もし二つのグラフ構造の併合のみに注目し，節点の値の扱いを無視するならば，手間はグラフの大きさに関してほぼ線形である．併合すべきグラフの大きさは，

[4]有理項とは，その部分項の集合が有限集合であるような，有限または無限の大きさの項である[9]．

プログラムで使用するデータ構造の**複雑さ** (データ構造の再帰的定義の大きさで測る) に依存するが，作られるデータ構造の**大きさ**には依存しない．たとえば，ストリームのストリームは，定数のストリームよりも複雑なデータ構造である．

モードグラフの大きさ l は，プログラムの大きさにも依存する．大きなプログラムほど，多くの述語記号を使うであろうからである．しかし，プログラムが扱う最も複雑なデータ構造の複雑さは，プログラムが大きくなるにしたがって拡大しないかもしれない．そこで，上のアルゴリズムの時間計算量を，もっと多くのパラメタを使って表現し直すことにしよう:

- 二つのモードグラフそれぞれに出現するトップレベルの素性 (つまり $\langle p, i\rangle$ ($p \in Pred$, $i \in n_p$) の形の素性) の数 w_1，w_2
- どちらのモードグラフにも出現するトップレベルの素性の数 c
- $\langle p, i\rangle$ の形のパス ($p \in Pred$, $i \in n_p$) にある節点を根とする部分モードグラフの最大の大きさ d

w_i はプログラムの大きさを反映し，d はデータ構造の最大の複雑さを反映することに注意してほしい．ハッシュ表 (乱アクセス用) とリスト構造 (順アクセス用) を併用してトップレベルの素性の集合を表現することにより，二つのモードグラフを併合する平均計算量は $O(\min(w_1, w_2) + cd{\cdot}\alpha(l))$ となる．項 $\min(w_1, w_2)$ はトップレベルの素性の集合を併合する手間を反映し，項 $cd{\cdot}\alpha(l)$ は共通のトップレベル素性をもつ部分モードグラフを併合する手間を反映する．項 $\alpha(l)$ は不可視ポインタをたぐる手間を反映するが，α はきわめてゆっくりとしか増大しないので，この項はほとんど定数と考えてよい．

モードグラフの併合と，素性グラフの単一化との相違点は，時間計算量には影響しない．まず，節点のラベルと辺の極性は，個々の辺をたぐるときに定数時間で扱うことができる．また，“基底” ラベルのついた節点の扱いが時間計算量を増大させることもない．なぜなら，1 個の “基底” 節点だけからなるグラフと他のグラフとの単一化は，後者のグラフを，不可視ポインタをたぐりつつ深さ優先探索することと本質的に同等であるが，これは上記の時間計算量で行なうことができるからである．

1.7.2 プログラム全体の解析

これで，プログラム全体のモード解析の時間計算量を考察する準備が整った．本節ではまだ，第 1.7 節の冒頭で述べた二つの仮定を行なっている．

まず最初に，プログラムが課する原始モード制約の総数は，記号の総数で測ったプログラムの大きさを n として，$O(n)$ であることに注意しておく．これは，

- 節の中の個々の関数記号または変数に対し，制約 (HF)，(HV)，(BF)，(BV) のそれぞれは高々1 個の原始モード制約しか課しないことと，
- 制約 (BU) はボディの単一化ゴールそれぞれに適用されること

から明らかである．また，個々の原始モード制約には，高々2 個のトップレベル素性しか出現しないことにも注意しておく．

二つのモードグラフの併合と同様，二つの側面を別々に考えよう．最初にこれらの $O(n)$ 個のトップレベル素性の集合の併合を考え，次に共通のトップレベル素性を持つ部分モードグラフの併合を考える．

まず，$O(n)$ 個のトップレベル素性の集合の併合の時間計算量は $O(n)$ である．なぜなら，大きさ w_1 および w_2 の二つの集合を併合する手間は $O(\min(w_1, w_2))$ であり，$\min(w_1, w_2) \leq 2$ だからである．しかし，大きなプログラムの解析では，個々のプログラム・モジュールのモードグラフを別個に求めておき，それらを後に**任意の**順序で併合したくなるかも知れない．インクリメンタルなプログラム開発では，大きなプログラムを毎回ゼロから解析したくないからである．トップレベル素性の集合を全部併合するのに，併合する二つの集合の大きさがいつもほとんど同じであるような順序で併合してゆくと，手間は $O(n \log n)$ になることがある．

次に，部分モードグラフの併合の手間を考える．あるトップレベル素性 $\langle p, i \rangle$ が，k 個の原始モード制約の中に出現するとしよう．すると，$\langle p, i \rangle$ にある節を根とする部分モードグラフの併合は，それらの原始モード制約の併合の順序がどうであっても，$k-1$ 回行なうこととなる．k は $O(n)$ であり，また高々二つのトップレベル素性が個々の原始モード制約の中に出現するだけなので，部分モードグラフの併合回数は全体で $O(n)$ 回である．個々の併合操作の時間計算量は，d を併合すべき最大の部分モードグラフの大きさとして，$O(d \cdot \alpha(n))$ である．項 $\alpha(n)$ は，本操作に関係してくる不可視ポインタの数が $O(n)$ である (証明略) ことを反映している．これらから，部分モードグラフの併合の手間は，全体で $O(nd \cdot \alpha(n))$ ということになる．

以上をまとめると，すべての原始モード制約を一度に併合するときは，解析の手間は $O(nd \cdot \alpha(n))$ であり，それらを任意の順序で併合するときは，手間は $O(n \log n + nd \cdot \alpha(n))$ である．しかし後者の場合でも，項 $n \log n$ の手間全体に対する寄与は，項 $nd \cdot \alpha(n)$ の寄与に比べてはるかに小さいと期待できる．

ここで議論しているプログラムのクラスについては，上の解析アルゴリズムの全正当性は，基礎となっている素性グラフの単一化アルゴリズムの全正当性 [1] からほとんど直ちに導くことができる．

1.7.3 一般の場合

では，ガードゴールをもつ節や，3 個以上のチャネル出現をもつ変数をもつ節はどのように扱えばよいであろうか？

節がガードゴールを持つ場合は，制約 (GV) も考えなければならない．ガードゴールの一般的かつ効率的な扱いについてはまだ十分検討していないが，実用上は，下記がすべて成り立つ場合を考えれば十分である:

- ガードゴールはすべて，組込みのテスト述語の呼出しである
- それらのテスト述語のモードグラフの大きさはそれぞれ $O(d)$ (したがって有限) である
- それらのモードグラフは，非テスト述語の解析の前に求まっている

これらの条件を満たす場合は，頭部の p とガードの p' に出現する変数 v に関する制約 (GV) は，

1. m/p' の部分モードグラフのコピーを作り，
2. 得られたコピーと，m/p の現在の部分モードグラフとを併合する

ことで実現できる．コピーを作るのは，制約がテスト述語へ伝播するのを防ぐためであるが，これは $O(d)$ 時間でできる．(テスト述語のモードグラフからは不可視ポインタをあらかじめ除いておくことができるから，不可視ポインタのたぐり操作はここでは不要である．) さらに，制約 (GV) を考慮しても，併合すべきトップレベル素性の数は相変わらず $O(n)$ である．したがって，制約 (GV) は，1.7.2 節で求めた時間計算量を増加させない．

ある変数が，節の中で 3 個以上のチャネル出現をもつ場合は，その変数が課する制約 (BV) を単一化で直ちに解くことはできない．しかし多くの場合，そのような制約は，他の制約の助けを借りて二項または単項の制約 (2 本ないしは 1 本のパスしか関係しない制約) に簡約化でき，その場合は生成と検査 (generate-and-test) による well-moding の探索は必要ない．実際には，ほとんどすべての場合がこれに該当すると我々は予想している．たとえば，同じ入力データを共有しあう複数の兄弟プロセス q を生成するための述語 p は，下記のような節をもつであろう:

```
p(X, ...) :- ... | q(X, ...), p(X, ...)
```

この節からは，述語 p のモードがそのすべての呼出しに共通であるという暗黙の制約を使って，直ちに $m/\langle \mathrm{q}, 1\rangle = IN$ を推論できる．一方，部分モード $m/\langle \mathrm{p}, 1\rangle$ の値はまったく制約されない．

そこで，制約を解く戦略としては，単項または二項の制約を最初に解き，そうでない制約の取扱いを遅延させるのが最善であるということになる．残念ながら，生成と検査による探索か，あるいは三項以上の制約を解く強力な簡約規則を要するようなプログラムもないわけではない．以下に示す，well-moded でない節集合がその一例である:

```
p1 :- true |       r(X), s(X), t(X)
p2 :- true | q(X),       s(X), t(X)
p3 :- true | q(X), r(X),       t(X)
p4 :- true | q(X), r(X), s(X)
```

しかしそのような強力な解法は，病的なプログラムを解くのにしか必要でなく，その有用性は実装の努力に見合わないであろう．それよりはるかに有望な行き方は，プログラマに，3 個以上のチャネル出現を持つ変数が出現するかもしれないパスの部分モードを，何らかの形で**宣言**してもらうことで，生成と検査による探索を完全に避けるという方策であろう．これは理にかなったものである．なぜなら，ごく少数の変数しか，節の中に 3 回以上チャネル出現することはなく，しかもそれらの変数は，ほとんど常に単純な情報の流れをもち，その宣言は容易であるからである．

1.8 理論的諸結果

本節では，まずゴール節の1ステップの簡約化—つまり非単一化ゴールをあるプログラム節のボディゴールで置き換える操作，または単一化ゴールの実行—に関する基本定理を示して証明する．

補題1 m を節 C の well-moding とし，$t_1 =_k t_2$ を C の中の単一化 (ボディ) ゴールとする．すると，(i) $m(\langle =_k, i\rangle) = out$ であり，かつ (ii) t_i が変数であるような i が存在する．

証明 (i) は制約 (BU) から直ちに導かれ，(ii) は (i) と制約 (BF) の対偶から直ちに導ける．(証明終)

v を変数，t を項とする．t が v 自身であるか，t が v を含むとき，v と t の間の単一化の**拡張出現検査** (extended occur check) が**失敗**するという．

定理1 m を，プログラム $\mathcal{P}$ とゴール節 G の well-moding とする．G が1ステップ簡約化されて，ゴール節 G' になったとすると，m は $\mathcal{P}$ と G' の well-moding でもある．ただし，ここで実行されたゴール $g \in G$ は，拡張出現検査に失敗する単一化ゴールではないとする．

証明 二つの場合がある．記法を簡略にするため，ここではゴール節と，そのゴール節がもつボディゴールのマルチ集合とを同一視する．

[場合1] この簡約化では，非単一化ゴール g を，"h :- ... | B" の形の節 $C \in \mathcal{P}$ (新しい変数で名前替えしたもの) で書き換えた．

GHC の同期の規則によれば，$g = h\theta$ であるような代入 θ があり，また，$G' = G \setminus \{g\} \cup B\theta$ である．ここで '$\setminus$' と '$\cup$' はそれぞれ，マルチ集合の差と和をとる演算である．

我々が検討しなければならないのは，$g \in G$ に出現する変数が課する制約 (BV) と，θ によって $B\theta$ ($\in G'$) に持ち込まれた関数の出現 (これらの出現は，この関数の g における出現に端を発するものである) が課する制約 (BF) である．G' 中の他の変数が課する制約と，G' 中の関数の他の出現 (それらはすでに，$G\setminus\{g\}$ か B の中にあったものである) が課する制約は，簡約化前とまったく同じであるから考えなくてよい．そこで，g の中の各記号，つまり $\tilde{g}(p) \neq \perp$ を満たすような $\tilde{g}(p)$ ($p \in P_{Atom}$) について考える:

A. $\tilde{g}(p)$ が関数 f である場合．この場合は，
  (i) $\tilde{h}(p) = f$ であるか，さもなければ
  (ii) $p = p'q$ であり，かつ $\tilde{h}(p')$ が変数 (v とする) であるような $p' \in P_{Atom}$ および $q \in P_{Term}$ がある．

  (i) の場合は，この出現は簡約化時に消えるので，制約 (BF) は適用されない．そこで (ii) の場合だけ考えればよい．(ii) では，f の出現が $B\theta$ に新たに持ち込まれる可能性がある．v が，B 中に n (≥ 0) 回，$r_1, \ldots, r_n$ に出現するとし，g_j を，r_j における出現が属するゴールとしよう．これは，G' において，$\widetilde{g_i\theta}(r_iq) = f$ ($i = 1, \ldots, n$) が成立することを意味する．m が G' の well-moding であるためには，制約 (BF) から，$m(r_iq) = in$ が成り立たなければならない．しかし，これは以下のように導く

ことができる:

(1) $m(p) = in$ (G 中の f に制約 (BF) を適用)

(2) $\mathcal{R}(\{\overline{m/p'}\} \cup \bigcup_{1 \le i \le n}\{m/r_i\})$ (C 中の v に制約 (BV) を適用)

(3) $\mathcal{R}(\{\overline{m/p}\} \cup \bigcup_{1 \le i \le n}\{m/r_i q\})$ ((2) と性質 (1.8) から)

(4) $m(r_i q) = in, \quad 1 \le i \le n$ ((1) と (3) から)

B. $\tilde{g}(p)$ が変数 w であり，w は g に l (≥ 1) 回，$p_1(=p), p_2, \ldots, p_l$ に出現し，$G \setminus \{g\}$ には m (≥ 0) 回，$p_{l+1}, \ldots, p_{l+m}$ に出現する場合．$g = h\theta$ であるような代入 θ があるので，各 p_i ($1 \le i \le l$) について，$p_i = p'_i q_i$ であり，かつ $\tilde{h}(p'_i)$ が変数 (v_i とする) であるようなパス $p'_i \in P_{Atom}$ および $q_i \in P_{Term}$ が存在する．v_i は B に n_i (≥ 0) 回，$r_{i1}, \ldots, r_{in_i}$ に出現するとする．このとき

(1) $\mathcal{R}(\bigcup_{1 \le i \le l+m}\{m/p_i\})$ (G 中の w に制約 (BV) を適用)

(2) $\mathcal{R}(\{\overline{m/p_i}\} \cup \bigcup_{1 \le j \le n_i}\{m/r_{ij}q_i\}), \quad 1 \le i \le l$
(C 中の v_i に適用した制約 (BV) と，性質 (1.8) から)

(3) $\mathcal{R}(\bigcup_{1 \le i \le l}\bigcup_{1 \le j \le n_i}\{m/r_{ij}q_i\} \cup \bigcup_{l+1 \le i \le l+m}\{m/p_i\})$
((1)，(2)，性質 (1.7) から)

ここでまず，どの v_i も，h 中に 2 回以上現れない場合を考えよう．この場合，w は $B\theta$ 中に $n_1 + \cdots + n_l$ 回，$r_{ij}q_i$ ($1 \le i \le l, 1 \le j \le n_i$) に現れるが，性質 (3) は，制約 (BV) を G' 中の w に適用したものにほかならない．
次に，ある v_k が h 中に 2 回以上出現する場合を考える．そのような k の集合を K とする．この場合，w の出現回数は $n_1 + \cdots + n_l$ より少ないかも知れない．g 中の w の二つの出現を，h 中の同じ変数の異なる出現が受け取ると，それらは $B\theta$ に独立には持ち込まれないからである．しかし，

(4) $m/p'_k = IN, \quad k \in K$ (C 中の v_k に制約 (HV) を適用)

(5) $\mathcal{R}(\{\overline{m/p'_k}\} \cup \bigcup_{1 \le j \le n_k}\{m/r_{kj}\}), \quad k \in K$
(C 中の v_k に制約 (BV) を適用)

(6) $m/r_{kj} = IN, \quad k \in K, 1 \le j \le n_k$ ((4)，(5)，性質 (1.4) から)

(7) $m/r_{kj}q_k = IN, \quad k \in K, 1 \le j \le n_k$ ((6) から)

これは，G' 中の w に適用する制約 (BV) は，v_k ($k \in K$) が $B\theta$ に持ち込む w の出現を無視してもよいことを意味する．なぜなら一般に，$\mathcal{R}(S \cup \{IN\}) \Leftrightarrow \mathcal{R}(S)$ が成り立つからである (性質 (1.3))．一方

(8) $\mathcal{R}(\bigcup_{1 \le i \le l, i \notin K}\bigcup_{1 \le j \le n_i}\{m/r_{ij}q_i\} \cup \bigcup_{l+1 \le i \le l+m}\{m/p_i\})$
((3)，(7)，性質 (1.3) から)

これは，G' 中の w に，v_k ($k \in K$) が持ち込んだ w の出現を無視して制約 (BV) を適用したものにほかならない．

[場合 2] この簡約化では，単一化ゴール $t_1 =_k t_2$ を実行した．補題 1 より，t_i が変数であり，$m(\langle =_k, i\rangle) = out$ であるような i が存在する．一般性を失うことなく，$i = 1$ である (つまり，単一化は常に，**左辺**の変数への代入である) と仮定してよい．拡張出現検査の仮定により，項 t_2 は，変数 t_1 自身や，それを含む項ではない．したがって $\theta = \{t_1 \leftarrow t_2\}$ とすると，$G' = (G \setminus \{t_1 =_k t_2\})\theta$ である．変数 t_1 が，G 中に他に n (≥ 0) 回，$r_1, \ldots, r_n$ に出現すると仮定しよう．以下では，t_2 中の記号，すなわち $\tilde{t_2}(q) \neq \perp$ であるような $\tilde{t_2}(q)$ $(q \in P_{Term})$ について考えてゆく:

A. $\tilde{t_2}(q)$ が関数記号 f の場合.

(1) $m(\langle =_k, 2\rangle q) = in$ (制約 (BF) から)

(2) $m(\langle =_k, 1\rangle q) = out$ ((1) と制約 (BU) から)

(3) $m(r_i q) = in, \quad 1 \leq i \leq n$ ((2) と，G 中の t_1 に適用した制約 (BV) から)

$t_1 =_k t_2$ を実行すると，f は G' 中に新たに，$r_1 q, \ldots, r_n q$ に出現するようになる．しかし上に示したように，m はこれらの出現が課する制約 (BF) を満たす.

B. $\tilde{t_2}(q)$ が変数 w $(\neq t_1)$ であり，w は $t_1 =_k t_2$ 中に l (≥ 1) 回，$p_1 (= \langle =_k, 2\rangle q), p_2, \ldots, p_l$ に出現し，$G \setminus \{t_1 =_k t_2\}$ には m (≥ 0) 回，$p_{l+1}, \ldots, p_{l+m}$ に出現する場合．q_i $(1 \leq i \leq l)$ を，$\langle =_k, 2\rangle q_i = p_i$ を満たすようなパスとする．すると

(1) $\mathcal{R}(\bigcup_{1 \leq i \leq l+m} \{m/p_i\})$ (G 中の w に制約 (BV) を適用)

(2) $\mathcal{R}(\{m/\langle =_k, 1\rangle q_i\} \cup \bigcup_{1 \leq j \leq n} \{m/r_j q_i\}), \quad 1 \leq i \leq l$
(G 中の t_1 に適用した制約 (BV) と，性質 (1.8) から)

(3) $m/\langle =_k, 1\rangle q_i = \overline{m/p_i}, \quad 1 \leq i \leq l$ ($=_k$ に制約 (BU) を適用)

(4) $\mathcal{R}(\bigcup_{1 \leq i \leq l} \bigcup_{1 \leq j \leq n} \{m/r_j q_i\} \cup \bigcup_{l+1 \leq j \leq l+m} \{m/p_j\})$
((1), (2), (3), 性質 (1.7) から)

しかし，性質 (4) は，制約 (BV) を G' 中の w に適用したものにほかならない．(証明終)

定理 1 は，実行したゴール g が $v = v$ の形のときは成り立たないことに注意してほしい．次の場合を考えてみよう:

G : `:- p(A,A), q(A)`

$\mathcal{P}$: {`p(X,Y) :- true | X=Y`}

プログラム $\mathcal{P}$ は $m/\langle \mathrm{p}, 1\rangle = \overline{m/\langle \mathrm{p}, 2\rangle}$ という制約を課するが，これとゴール節 G の変数 `A` に適用した制約 (BV) とを組み合わせると $m/\langle \mathrm{q}, 1\rangle = IN$ を得る．しかし，`p(A,A)` を実行し，ついで生成したサブゴール `A=A` を実行すれば，ゴール節は

```
:- q(A)
```

となり，これは制約 $m/\langle \mathrm{q}, 1\rangle = IN$ に反する．ここでの問題点は，$v = v$ の形のゴールを実行すると，ほかの v の出現へ値を供給することなく，v の出力出現が消えてしまうことにある．幸いにして，$v = v$ の形の単一化ゴールは，いかなる実用的なプログラムでも使

うことはない．さらに，出現検査 (が実装されているならば，それ) を少し拡張するだけでこれらのゴールを容易に検出することができる．

もし，変数が節の中に 3 回以上チャネル出現することを禁止するならば，$v = v$ の形のゴールを認めても定理 1 が成立する．しかし，この代案は下の定理 2 ではうまくゆかない．

定理 1 と，ゴール節に適用した制約 (BV) とから，well-moded なプログラムは，拡張出現検査に失敗しない限り，第 1.3 節の冒頭に示したプログラミング・スタイルを守っていることが保証される．

第 1.5 節に，制約 (BV) の弱い版を示したが，これを用いても，定理 1 への影響はない．

補題 1 と定理 1 から，(簡約化によって導いた) ゴール節中の単一化ゴールは，つねに変数と項との単一化であることがわかり，そこから下の重要な系が導ける:

系 1 プログラムとゴール節の対が well-moded であり，拡張出現検査が失敗しないならば，この対は単一化 (ボディゴール) の失敗を引き起こすことはない．

$v = v$ の形の単一化ゴールを排除すると，もう一つ強力な定理が得られる:

定理 2 m をプログラム $\mathcal{P}$ とゴール節 G の well-moding とする．G の実行が，拡張出現検査の失敗を引き起こすことなく成功した (つまり G が，ゴールの空のマルチ集合に簡約化できた) とすると，その実行中に，G 中のどの変数 v に対しても，$v =_k t$ の形 ($m(\langle =_k, 1\rangle) = out$)，または $t =_k v$ の形 ($m(\langle =_k, 2\rangle) = out$) の単一化ゴールが実行されたはずである．

証明 制約 (BV) から，v は G 中にちょうど 1 個の出力出現をもつ．その出現をもつゴールを g とし，p を，$\tilde{g}(p) = v$ かつ $m(p) = out$ を満たすようなパスとしよう．一般性を失うことなく，$\mathcal{P}$ および G 中のすべての単一化ゴールに対して，$m(\langle =_k, 1\rangle) = out$ が成り立つと仮定してよい．G が空節ならば定理は明らかに成り立つので，以下では G が空でなくて成功したと仮定する．G の最初の簡約化によってゴール節 G' が得られたとすると，この簡約化は次の四つの場合に分類できる:

(i) ゴール g が $v =_k t$ の形 (つまり $p = \langle =_k, 1\rangle$) で，この簡約化では g を実行した．

(ii) ゴール g が $w =_k t$ の形で，t が v の出力出現をもち (つまりある $q \in P_{Term}$ に対して $p = \langle =_k, 2\rangle q$ が成り立ち)，この簡約化では g を実行した．w が $G \setminus \{w =_k t\}$ 中に n (≥ 0) 回，$r_1, \ldots, r_n$ に出現するとしよう．すると，

(1) $m(\langle =_k, 2\rangle q) = out$ (仮定から)

(2) $m(\langle =_k, 1\rangle q) = in$ ((1) と制約 (BU) から)

(3) $\mathcal{R}(\{m/\langle =_k, 1\rangle, m/r_1, \ldots, m/r_n\})$ (G 中の w に制約 (BV) を適用)

(4) $\exists k \leq n(m(r_k q) = out)$ ((2) と (3) から)

(5) $n > 0$ ((4) から)

w のこれらの n 個の出現は，簡約によって t に書き換えられるので，G' には少なくとも 1 回，v が出現する．

(iii) ゴール g は非単一化ゴールで，この簡約化では g を節 "h :- ... | B" を用いて簡約化した．このときは，$p = p'q$ でしかも $\tilde{h}(p')$ が変数 (w とする) であるようなパス $p' \in P_{Atom}$ および $q \in P_{Term}$ があるはずである．w が B に n (≥ 0) 回，$r_1, \ldots, r_n$ に出現するとしよう．すると

(1) $m(p'q) = out$ (仮定から)
(2) $\mathcal{R}(\{\overline{m/p'}, m/r_1, \ldots, m/r_n\})$ ("h :- ... | B" 中の w に制約 (BV) を適用)
(3) $\exists k \leq n(m(r_k q) = out)$ ((1) と (2) から)
(4) $n > 0$ ((3) から)

v は，G' では新たに $r_1 q, \ldots, r_n q$ に出現するので，G' には少なくとも 1 回，v が出現する．

(iv) この簡約化では，g 以外のゴール g' を実行した．v の出力出現をもつのは g なので，g' はそれをもたないことに注意してほしい．この簡約化では変数 v は書き換えられることがない．なぜなら，v の出力出現をもつ，$v =_k t$ の形の単一化ゴールのみが，v を書き換えることができるからである．したがって G' は g の具体形を含み，それは v の出力出現を含む．

これを要約すると，G から G' への簡約化では，

(a) $v =_k t$ の形の単一化ゴールが実行されたか，または
(b) G' が v の出力出現を保った

ことになる．m は G' の well-moding でもあるので，(b) の場合，上の議論は G' の簡約化に対しても適用できる．しかし仮定より，G は最終的にはゴールの空のマルチ集合に簡約化されている．したがって (a) の場合がいつかは起きたはずである． (証明終)

定理 2 から，下記の系が導ける:

系 2 定理 2 の仮定の下で，単一化 (ボディ) ゴールが生成したすべての代入の積は，G 中のすべての変数を基底項に書き換える．

証明 ここでも，$\mathcal{P}$ 中のすべての単一化ゴールについて $m(\langle =_k, 1\rangle) = out$ と仮定する．定理 2 から，G 中に出現するどの変数 v についても，$v =_k t$ の形のゴール ($m(\langle =_k, 1\rangle) = out$) が実行され，$v$ が t に書き換えられたはずである．これを二つの場合に分類する:

(a) t が基底項のとき．この場合はこの系は直ちに成り立つ．
(b) t が 1 個以上の変数 $v_1, \ldots, v_n$ を含む．ゴール $v =_k t$ を実行したのは，ゴール節 G' の簡約化のときであったとしよう．(一番最初に実行したのが $v =_k t$ であったとき，そしてそのときのみ，G' は G 自身である．) 定理 1 から，m は G' の well-moding である．したがって，この系が $v_1, \ldots, v_n$ および G' に対して成り立つならば，v および G に対しても成り立つ．

しかし，G の実行に成功したときは，その中で高々有限個の単一化ゴールしか実行しなかったのであるから，(b) における再帰が無限回起きることはありえない．したがって，

単一化ゴールを実行することによって生成したすべての代入の積は，最終的に v を基底項に書き換えたはずである． (証明終)

このように，拡張出現検査の下では，停止性から，変数値の基底性が帰結できる．

下のような節の対が，系2の反例になっているのではないかと思う人がいるかもしれない:

```
:- p(A)
p(X) :- true | X=f(Y)
```

しかし我々のモード体系は，このような対を許容していない．なぜならゴール節が制約 $m/\langle \mathtt{p},1\rangle = OUT$ を課する一方で，プログラム節は制約 $m/\langle \mathtt{p},1\rangle\langle \mathtt{f},1\rangle = IN$ を課するからである．上の例を well-moded にするには，たとえば以下のように，Y を具体化するゴールを与えなければならない:

```
:- p(A), q(A)
p(X) :- true | X=f(Y)
q(f(Y)) :- true | Y=g
```

1.9 KL1への適用

提案するモード解析を，現存の KL1 プログラムに適用する場合，次のような機能の扱いを検討する必要がある:

1. 出力引数をもつガードゴール
2. 算術演算
3. 単一化不可能性の検査
4. 節の順序づけ機能とガード部の逐次実行
5. functor，arg 等による項の操作
6. ベクタやストリングとその操作
7. 荘園
8. 入出力
9. 高階機能 (データとして存在する述語コードの，引数への適用)

これらのうち，高階機能を除く各機能について，順に検討してゆく．高階機能については第2.5.3節で論じる．

なお本節では，ガードで呼び出す述語の課するモード制約はすべて既知であり，単一化ボディゴールと同様に，個々の呼出しのモード (呼出し側の制約と述語の制約からきまる) は区別して考えるものとする．

1.9.1 出力引数をもつガードゴール

Flat GHC では，プログラムの意味論を単純化し，解析や変換を容易にするために，ガードゴールをテスト述語の呼出しに限定し，ガードゴールがその引数を具体化する可能性

を構文的に排除している．しかし現在のKL1では，引数を具体化するガードゴールをいくつか認めているので，それらの解析手法について考える．

引数を具体化するガードゴールを認めるということは，非テスト述語の呼出しを認めるということである．しかし，

- 相異なる変数を各出力引数に与えて呼び出す限り，成功するか否か，および出力引数の値がどうなるかが，入力引数から一意に決まる
- ガードゴール間のデータ依存性(モード解析でわかる)が循環しておらず，それらをあらかじめ決めた順序で逐次的に実行できる

の両条件を満たせば，理論上および実用上，Flat GHCの設計指針を大きく逸脱するものではない．しかし，引数を具体化するガードゴールがあると，それらのガードゴールが非協調的に変数を具体化する場合を扱わなければならないという困難が生じる．これまでに考えたモード体系は，変数の具体化が協調的であるという仮定に基づいていたが，その仮定を変えなければならなくなるわけである．

たとえば，arg2(t,t_1,t_2)を，2引数の項tの第1引数をt_1と，第2引数をt_2と単一化するゴールとしよう．arg2は

$$\texttt{arg2(}f\texttt{(X1,X2),Y1,Y2) :- true | Y1=X1, Y2=X2}$$

の形の節($f \in Fun$)の集まりで定義してあると考えられ，したがってこの述語からは

$$m/\langle \texttt{arg2},1\rangle\langle f,1\rangle = \overline{m/\langle \texttt{arg2},2\rangle}, \quad f \in Fun \tag{1.9}$$
$$m/\langle \texttt{arg2},1\rangle\langle f,2\rangle = \overline{m/\langle \texttt{arg2},3\rangle}, \quad f \in Fun \tag{1.10}$$

という制約がかかる．

このとき，節

$$\texttt{p(X,Y) :- arg2}_1\texttt{(X,T,\_), arg2}_2\texttt{(Y,\_,T) | ...}$$

のガードは，Xの第1引数とYの第2引数が同じ値かどうかを検査しており，したがって

$$m/\langle \texttt{p},1\rangle\langle f,1\rangle = m/\langle \texttt{p},2\rangle\langle g,2\rangle = IN, \quad f,g \in Fun \tag{1.11}$$

であるべきである．しかるに，(1.9) ~ (1.11)と

$$m/\langle \texttt{p},1\rangle = m/\langle \texttt{arg2}_1,1\rangle \quad (\texttt{X}\text{に関する協調性}) \tag{1.12}$$
$$m/\langle \texttt{p},2\rangle = m/\langle \texttt{arg2}_2,1\rangle \quad (\texttt{Y}\text{に関する協調性}) \tag{1.13}$$
$$m/\langle \texttt{arg2}_1,2\rangle = \overline{m/\langle \texttt{arg2}_2,3\rangle} \quad (\texttt{T}\text{に関する協調性}) \tag{1.14}$$

をすべて満たすようなモードmは存在しない．たとえば(1.12)と(1.13)を満たすならば，(1.9) ~ (1.13)から

$$m/\langle \texttt{arg2}_1,2\rangle = m/\langle \texttt{arg2}_2,3\rangle = OUT \tag{1.15}$$

が導けるからである．また，

$$\texttt{q(X) :- arg2}_3\texttt{(X,h(Y),\_) | ...}$$

のガードも，

$$m(\langle \mathtt{q},1\rangle\langle \mathtt{f},1\rangle) = in, \quad f \in Fun \tag{1.16}$$

と解釈すべきであるが，制約 (1.9) と

$$m(\langle \mathtt{arg2}_3,2\rangle) = in \quad (\text{(BF) をガードゴールに準用}) \tag{1.17}$$

からは

$$m(\langle \mathtt{arg2}_3,1\rangle\langle \mathtt{f},1\rangle) = out, \quad f \in Fun \tag{1.18}$$

が導かれ，これは (1.16) と

$$m/\langle \mathtt{q},1\rangle = m/\langle \mathtt{arg2}_3,1\rangle \quad (\mathtt{X} \text{に関する協調性}) \tag{1.19}$$

から導ける

$$m(\langle \mathtt{arg2}_3,1\rangle\langle \mathtt{f},1\rangle) = in, \quad f \in Fun \tag{1.20}$$

と矛盾する．

その一方で，節

$$\texttt{r(X) :- arg2(X,T,\_), T>5 | ...}$$

のように，すべての変数の具体化が協調的であるという仮定でモードづけすべき場合も存在する．つまり，制約 (BV) を適当に拡張してガードゴールに適用できそうな場合と，そうでない場合とがあるわけである．

以上の問題を整理するために，かりに，制約 (BV) を下記のように変更したとしよう:

(GBV) 変数 v が，h，G，B の中にちょうど $n\,(\geq 1)$ 回，$p_1,\ldots,p_n$ に出現し，そのうち h 中の出現が $p_1,\ldots,p_k$ $(k \geq 0)$ であるとする．このとき

$$\mathcal{R}(\{\overline{m/p_1},\ldots,\overline{m/p_k},m/p_{k+1},\ldots,m/p_n\})$$

このような規則を適用すると，上記の述語 p，q のように，ガードゴールの簡約によって生成する単一化ゴールの実行が，頭部から受け取った値の検査となる場合に不都合が生じる．また，

$$\texttt{r(X,X) :- true | ...}$$

のように，頭部に同一変数が複数回出現する節においても不都合が生じる．

そこで，これらの場合を静的に検出することを考えてみよう．まず，次のような性質を満たす $P_{Atom} \cup \{fun\}$ 上の同値関係 '$\rightleftharpoons$' (反射律，交換律，推移律が成り立つ関係) を考える:

1. ある変数が，同じ節のガード部の p_1 と p_2 とに出現する場合，$p_1 \rightleftharpoons p_2$
2. ガードゴール $g \in G$ の簡約によって生成しうる単一化ゴールが，g 中のパス p_1 と p_2 とを単一化する場合，$p_1 \rightleftharpoons p_2$
3. ガードゴール $g \in G$ の簡約によって生成しうる単一化ゴールが，g 中のパス p を非変数値に具体化する場合，$p \rightleftharpoons \mathit{fun}$

たとえば上記の述語 p の例では，

$$\begin{aligned}\langle \mathrm{p},1\rangle\langle f,1\rangle &\rightleftharpoons \langle \mathrm{arg2}_1,1\rangle\langle f,1\rangle \rightleftharpoons \langle \mathrm{arg2}_1,2\rangle \rightleftharpoons \langle \mathrm{arg2}_2,3\rangle \\ &\rightleftharpoons \langle \mathrm{arg2}_2,1\rangle\langle g,2\rangle \rightleftharpoons \langle \mathrm{p},2\rangle\langle g,2\rangle\end{aligned}$$

となる．直観的には，関係 '$\rightleftharpoons$' は，二つのパスどうし，またはあるパスと関数記号とが，直接間接に「つながるかも知れない」ことを表わす．関係 '$\rightleftharpoons$' を用いると，上記の不都合は，節の頭部のパス p, p' について，$p \rightleftharpoons p'$ または $p \rightleftharpoons \mathit{fun}$ が成り立つ場合に発生するということができる．

このように，節の頭部のパス p, p' について $p \rightleftharpoons p'$ または $p \rightleftharpoons \mathit{fun}$ である場合に対しては，制約 (GBV) に優先する特別なモードづけの規則を定めるのも一つの方策である．が，モードづけの規則をできるだけ単純化し，例外規定を避けるには，$p \rightleftharpoons p'$ または $p \rightleftharpoons \mathit{fun}$ であるようなパスを頭部にもつ節を禁止してしまうほうがよかろう．つまり，頭部で受け取った値の検査は，頭部に明示した関数記号で行なう場合以外は，すべて検査のための述語を呼び出して行なうようにするわけである．

具体的には，両辺がまったく同一であることを検査する組込述語 ('==' とする)，および両辺の主関数記号が同一であることを検査したのち，その対応する引数どうしを単一化する組込述語 ('?=' とする) を用意し，述語 p，q，r の例はそれぞれ

```
p(X,Y) :- arg2_1(X,T1,_), arg2_2(Y,_,T2), T1==T2 | ...
q(X) :- arg2_1(X,T1,_), T1?=h(Y) | ...
r(X1,X2) :- X1==X2 | ...
```

と書くようにすればよい．

こうすれば，等価性の判断が，組込述語 '==' および '?=' のガードの実行として明示的に行なわれるので，プログラマの意図が明確になるし，各変数への値の供給源もそれぞれ一つになる．このとき，モード制約規則は次のように変化する:

- 制約 (BV) に代えて，(GBV) を採用する
- 制約 (GV) も，「ガードで呼び出す述語のモードは既知」という前提の下では，(第 1.5 節で述べた制約伝播の問題に配慮しなくてよいので) (GBV) に吸収できる
- 頭部に同一変数が複数回出現するのを禁止するため，制約 (HV) が不要になる

ただし，(BV) に付随して定義したチャネル出現の概念を修正しなければならない．特に，テスト述語の呼出しに出現する変数については，まったくモード制約が課されないことがあるので，さらに検討を進める必要がある．また，第 1.5 節で (BV) の弱い版を考えたが，それに対応するものは，(GBV) に対しても考えることができよう．

なお，上記の変更はあくまでも，$p \rightleftharpoons p'$ または $p \rightleftharpoons \mathit{fun}$ であるようなパス p, p' が頭部中にないことの検査を前提としていることに注意してほしい．

1.9.2 算術演算

整数や浮動小数点数の演算を簡潔に記述するために，KL1では，':='のような代入述語の右辺や，'=:='のような比較述語の両辺に算術式を書くことを認めている．

KL1ではこのような述語をマクロとして定義しているので，マクロ展開の後にモード解析をすることも考えられる．しかし，これらの述語がマクロで定義されているか否かは，処理系作成技法に関わることがらであるから，マクロ展開前のプログラムそのものに対してもモードづけができることが望ましい．

ここで問題となるのは，変数を含む算術式に関する言語仕様である．以下では':='を例にとって検討するが，算術式を引数にとる他の述語でも同じ議論が成り立つ．

算術式はいくらでも複雑でありうるので，すべての':='の呼出しに対して同一のモードを与えようとすると，$m/\langle :=, 2\rangle = IN$としなければならない．これは，`2*X+1`のような算術式の中の変数`X`が実行時に算術式に具体化することを認めるならば，合理的な制約である．しかし，`X`が整数値に具体化することしか認めないのであれば，上記のモード制約は必要以上に強すぎることとなる．このことのプログラミングへの影響は，実際にはそれほど大きくないと予想されるが，より現実的で合理的な方法は，':='については個々の呼出しごとに，述語自身が課するモード制約を引数に応じて変えることであろう[5]．つまり，$v :=_k e$の形のゴールがあり，eの中に変数がn回，$\langle :=_k, 2\rangle q_1, \ldots, \langle :=_k, 2\rangle q_n$に出現する場合，'$:=_k$'は

$$m(\langle :=_k, 2\rangle q_i) = in, \quad 1 \leq i \leq n$$

および

$$m(\langle :=_k, 1\rangle) = out$$

という制約のみを課することにするわけである．

1.9.3 単一化不可能性の検査

KL1では，「二つの項が永遠に単一化できない」ことを表わす組込述語'\='を用意している．たとえば，

```
p(X) :- X \= f(Y) | ...
```

のガードは，「pの呼出しの引数Xの主関数記号がfでない」ことを検査しており，これはちょうど

```
p'(Y) :- X = f(Y) | ...
```

のガード条件の否定になっている．また，

```
q(X,Y) :- X \= f(Y) | ...
```

のガードは，「pの第1引数の主関数記号がfであり，かつfの第1引数の値がpの第2引数と同一である」ことはないことの検査をしていて，同様に

```
q'(X,Y) :- X = f(Y) | ...
```

[5]単一化述語'='の場合は，異なる呼出しが異なるモードを持てるようにしただけであり，個々の述語'$=_k$'が課する制約(BU)自身はすべて同じである．

のガード条件の否定になっている．

述語 q の例からは，述語 '\=' にモードを与えるとすれば

$$m/\langle \backslash =, 1\rangle = m/\langle \backslash =, 2\rangle = IN$$

とすべきであることになる．述語 '\=' を定義する節集合を仮想的に考えても，同じことが言える．しかし，これと制約 (GV) とを組み合わせると，述語 p に対して制約 $m/\langle$p, $1\rangle = IN$ が課されることになる．これは検査する必要のないパスのモードを制約しており，不当に強すぎる制約である．述語 p のガードが課する制約として合理的なのは，$m(\langle$p, $1\rangle) = in$ であり，これより強いものは不都合である．

一つの方向は，':=' の場合と同様，'\=' についても，引数に応じて述語自身のモード制約を変えることである．しかし，':=' の場合と異なり，合理的なモード制約は，プログラム中に書かれた '\=' の引数自身だけからは決めることができない．下記のいくつかの例を考えてみてほしい:

```
p(X)   :- X\=f(Y) | ...
q(X,Y) :- X\=f(Y) | ...
r(X,Y) :- X\=f(A), Y\=g(A) | ...
s(X,Y) :- arg2(X,A,_), Y\=f(A) | ...
t(X,Y) :- A:=X+1, Y\=f(A) | ...
```

詳説はしないが，これらの例からわかることは，'\=' の引数の中の変数がガード中に他に出現するか否か，どのように出現するかによって，'\=' 自身の課すべきモードが変わってくるということである．

以上の考察から，'\=' の機能は，モード解析の観点からは強力すぎると結論せざるを得ない．'\=' の機能は，効率的実装の観点からも強力すぎるという指摘があり，再検討の必要がある．

そこで，単一化不可能性の検査のためには，述語 '\=' に代えて，より基本的な機能を提供することを提案したい．具体的には，両引数の主関数記号が異なるときに成功する述語 ('\?=' とする) を導入するわけである．こうすれば，述語 p は，

```
p(X) :- X \?= f(Y) | ...
```

または

```
p(X) :- X \?= f(0) | ...
```

と書くことができる．'\?=' の課する制約は，

$$m(\langle \backslash ?=, 1\rangle) = m(\langle \backslash ?=, 2\rangle) = in$$

だけであるので，p に強すぎる制約が伝播することはなくなる[6]．

[6]KLIC 第 3 版の '\=' は，本検討の結果を反映してそのような仕様になっている．

1.9.4 節の順序づけ機能とガード部の逐次実行

KL1では，述語を定義する節に二通りの方法で順序を導入することができる．一つは指示語otherwiseである．あるゴールを簡約化するのに，述語定義の中で指示語otherwiseの後ろに書いた節は，前に書いた節を用いた簡約化が**永久に**不可能な場合にのみ試される．もう一つは指示語alternativelyである．指示語alternativelyの後ろに書いた節は，前に書いた節を用いた簡約化が**直ちには**不可能な場合にのみ試される．

これらの指示語は，モード解析のときには単に無視してよい．なぜなら，これらの指示語自身が新たなモード制約を課することはないし，指示語があってもなくても，指示語のあとに続く節が課するモード制約は変わらないからである．

指示語otherwiseの後ろに書いた節は，前に書いた節のガードが指定する条件の否定を，暗黙の条件としてもっていると考えることができる．しかし，ある条件の成立を調べるために検査するかもしれないパスの集合と，その条件の否定の成立を調べるために検査するかもしれないパスの集合は同一であり，otherwiseの存在によってモード制約が増減することはない．

指示語alternativelyは，優先度指定のための指示語であり，モード解析のような，プログラムの実行方法によらない性質の推論には本質的に影響しない．

KL1では，実装上の理由により，ゴールと節頭部とのマッチングを左の引数から順々にとり，またガードゴールはそれに引き続いて左から右へと実行することにしている．つまり，節のガード部は逐次実行している．このような実行法は，Flat GHCではまったく問題ない．Flat GHCはガード部の逐次実行を許すように設計してある．

しかしKL1には指示語otherwiseがあるため，節のガード部を逐次的に実行した場合と並行実行した場合とでは，プログラムの挙動が変わってくることがある．並行実行した場合に検出される失敗が，逐次実行では中断のために永遠に検出されず，otherwiseの後ろに書いた節が利用されなくなることがあるのである．またKL1ではガードで非テスト述語が呼び出せるが，それらが適切な順序で並んでいないと，並行実行では成功するものが，永遠に中断してしまうことがある．

これらのことは，モード解析の方法に影響するものではない．しかし，節のガード部の逐次実行を前提とするならば，otherwiseのような「デフォールトの場合」を指定する機能は，もっと制限した形で導入する方がよいであろう．つまり，節のガード部を逐次実行するか並行実行するかによって，プログラムの挙動が変化しないような形で導入することが望ましい．

その一つの方法は，otherwiseを，ゴール中のあるパスの値(関数記号)による場合分けにしか使えないようにするものである．たとえば，ストリームの先頭要素による場合分けは，図1.7のように行なうことが考えられる。

図1.7の場合1～3のように，Cが定数の場合については，1個の条件の中に複数の値を併記することを認め，ボディの中でCを使用できるようにするのが好都合であろう．また，Cが非定数項の場合については併記は認めず，その引数を相異なる変数(上例ではX)で受け取り，->の右辺ではCでなくそれらを使うようにすると，モードづけの観点からも好都合である．デフォールトの場合については，もちろんボディの中で変数Cへのアクセスを許す必要がある．上記のような形式に限定すると，場合の数が多いときはインデクシングが使え，ある程度効率のよい実装が保証できるという利点もある．

```
p([C|Cs], 残りの引数) :-
    switch(C,(
        9,10,32  -> ボディ1;  % 場合1
        48..57   -> ボディ2;  % 場合2
        65..90   -> ボディ3;  % 場合3
        error(X) -> ボディ4;  % 場合4
    otherwise;
                    ボディ5 ))
```

図 1.7: ストリームの先頭要素による場合分け

このように，一回のコミット操作のために行なう判断は，できるだけ簡単な形にし，複雑な枝分かれは複数回のコミット操作にわけて行なうようなプログラミングを推奨すべきであろう．

1.9.5 functor，arg 等による項の操作

KL1 には，項の主関数記号を調べるための述語 functor，ある主関数記号をもった項を作るための述語 new_functor，項の特定の引数にアクセスするための述語 arg，ある項と特定の引数の値だけが異なる項を作るための述語 setarg がある．

これらの組込述語を用意しても，それらを使用しないプログラムに不必要なモード制約が新たに加わることはない．使用した場合に，使った組込述語からのモード制約がかかるだけである．

一般に，組込述語のモードを考えるには，それらを定義する仮想的な節集合を考え，それが与えるモード制約を考えればよい．たとえば述語 arg の場合は，次のような節が，1 引数の f や 2 引数の f をはじめとするすべての関数記号 (引数の個数が異なれば，異なる関数記号である) に対して用意されていると考えることができる:

```
arg(1,f(A),X) :- true | X=A
arg(1,f(A,B),X) :- true | X=A
arg(2,f(A,B),X) :- true | X=B
...
```

この述語は，

$$m/\langle \mathtt{arg}, 2\rangle = IN, \quad m/\langle \mathtt{arg}, 3\rangle = OUT$$

という強い制約を課する．第 2 引数の複合項のすべての引数が，第 3 引数と単一化される可能性があるため同一の部分モードをもたなければならない上，第 2 節の B，第 3 節の A のように 1 度しかチャネル出現しない変数が強い制約を課するからである．これでも基底項の表の表引きに使うには不都合がないが，複合項をもっと柔軟に使いたいこと

もあろう．述語 setarg についても，

```
setarg(1,f(A),X,T) :- true | T=f(X)
setarg(1,f(A,B),X,T) :- true | T=f(X,B)
setarg(2,f(A,B),X,T) :- true | T=f(A,X)
```

等からモード制約を求めると，arg の場合と同様の理由で

$$\begin{aligned} m/\langle \texttt{setarg}, 2\rangle &= m/\langle \texttt{setarg}, 3\rangle = IN \\ m/\langle \texttt{setarg}, 4\rangle &= OUT \end{aligned}$$

という制約があることがわかる．データ構造の中に，捨てる可能性のある部分があると，強い制約がかかるのである．

これらの述語のモード制約を弱くするには，「データ構造を操作したあとは，すべてのデータを捨てずに呼出し側へ返す」ようにすればよい．これには，上記の arg や setarg に代えて，下記のような節で定義した，5引数の setarg を用意すればよい[7]:

```
setarg(1,f(A),X0,X,T) :- true | X0=A, T=f(X)
setarg(1,f(A,B),X0,X,T) :- true | X0=A, T=f(X,B)
setarg(2,f(A,B),X0,X,T) :- true | X0=B, T=f(A,X)
...
```

この setarg に対してかかるモード制約は，

$$\begin{aligned} m(\langle \texttt{setarg}, 1\rangle) &= in \\ m(\langle \texttt{setarg}, 2\rangle) &= in \\ m/\langle \texttt{setarg}, 2\rangle &= \overline{m/\langle \texttt{setarg}, 5\rangle} \\ m/\langle \texttt{setarg}, 2\rangle\langle f, i\rangle &= m/\langle \texttt{setarg}, 4\rangle, \quad f \in Fun,\ i \in N_f \\ m/\langle \texttt{setarg}, 3\rangle &= \overline{m/\langle \texttt{setarg}, 4\rangle} \end{aligned}$$

がすべてであり，arg や4引数の setarg に比べれば弱い．また，arg や4引数の setarg は，この5引数の setarg を用いて

```
arg(K,T,X) :- true | setarg(K,T,X,0,_)
setarg(K,T,X,T) :- true | setarg(K,T,_,X,T)
```

と容易に定義できるので，5引数の setarg を，複合項の引数へのアクセスの基本操作と考え，arg や4引数の setarg は基底複合項の操作のための変種とみなすのが適当である．上の4引数 setarg の定義では，X に置き換えられる要素を単に捨てているが，その要素が基底項でない場合は，捨てる前に何らかのターミネーション処理をする必要があるのである．

次に，述語 functor は，

```
functor(f(X,Y,Z,W),F,N) :- true | F=f, N=4
```

[7]KLIC 第3版の setarg は，本検討の結果を反映してそのような仕様になっている．

のような節の集合で定義してあると考えてよい．したがって，

$$\begin{aligned} m/\langle \texttt{functor}, 1\rangle &= IN \\ m(\langle \texttt{functor}, 2\rangle) &= m(\langle \texttt{functor}, 3\rangle) = out \end{aligned}$$

となる．制約 $m/\langle \texttt{functor}, 1\rangle = IN$ は強いものであるが，functor の主要な用途は，

```
p(T) :- true | functor(T,F,N), p_args(T,N)
```

のように，項 T の各引数についての何らかの処理を行なう前に，その引数個数を調べることにある．このような節には T は 3 回出現し，しかも $m/\langle \texttt{functor}, 1\rangle = IN$ であるので，$m/\langle \texttt{p}, 1\rangle$ と $m/\langle \texttt{p\_args}, 1\rangle$ には，$m/\langle \texttt{p}, 1\rangle = m/\langle \texttt{p\_args}, 1\rangle$ という制約しかかからない．これは意図した制約と同じである．

述語 new_functor は，

```
new_functor(T,f,4) :- true | T=f(X,Y,Z,W)
```

のような節の集合で定義してあると考えることができる[8]．この節は，

$$\begin{aligned} m(\langle \texttt{new\_functor}, 1\rangle) &= out \\ m(\langle \texttt{new\_functor}, 2\rangle) &= m(\langle \texttt{new\_functor}, 3\rangle) = in \end{aligned}$$

のほかに，制約 (BF)，(BU)，(BV) から

$$m/\langle \texttt{new\_functor}, 1\rangle\langle \texttt{f}, i\rangle = IN, \quad 1 \leq i \leq 4$$

という強い制約を課する．これは都合の悪い制約である．

new_functor で作成する複合項の引数は，arg で具体化するのがもっとも普通の用法であろう．実際，制約 (BV) から，これらの引数である変数には，必ず値の供給源がなければならない．しかし，述語 arg のモード制約は

$$m/\langle \texttt{arg}, 2\rangle = IN, \quad m/\langle \texttt{arg}, 3\rangle = OUT$$

であったから，

```
new_functor(T,f,4),
    arg1(1,T,a), arg2(2,T,b), arg3(3,T,c), arg4(4,T,d)
```

というような使い方ができないことになる．

では，5 引数の setarg を使ったらどうなるであろうか？

```
new_functor(T,f,4),
    setarg1(1,T,a,A,T1), setarg2(2,T1,b,B,T2),
    setarg1(3,T2,c,C,T3), setarg2(4,T3,d,D,T4)
```

[8] これは new_functor の当初の仕様である．KLIC 第 3 版の new_functor は，本検討の結果を反映してそのような仕様になっている．

とすることにより，Tの引数を具体化してゆける．しかし，5引数のsetargを使うと，Tが具体化するだけでなく，最後に，新たな複合項T4 (= f(A,B,C,D))ができてしまう．この複合項が返ってくる$\mathtt{setarg}_4$の第5引数には，変数T1，T2，T3に関する制約 (BV) と，述語setargに関する制約

$$m/\langle \mathtt{setarg}, 2\rangle = \overline{m/\langle \mathtt{setarg}, 5\rangle}$$

とから，new_functorの第1引数と同じモード制約，つまり

$$m/\langle \mathtt{setarg}_4, 5\rangle\langle \mathtt{f}, i\rangle = IN, \quad 1 \leq i \leq 4$$

がかかる．要するに，5引数のsetargを使うと，もとの複合項は具体化できるものの，具体化しなければならない新たな複合項が作られてしまうわけである．これは，5引数のsetargも，new_functorで作成した複合項の具体化には不都合であることを意味している．

そこで，より弱いモード制約をもつnew_functorの変種を考えてみよう．まず，作成する複合項の各引数を，定数 (たとえば0) で初期化してしまうとどうなるであろうか? このような機能をもつ述語をnew_functor_outと名づける．別の値を引数にもつ複合項を作成したいときは，まずnew_functor_outで項を作成してから，setargで引数値の一部異なる項を作成すればよい．new_functor_outは

```
new_functor_out(T,f,4) :- true | T=f(0,0,0,0)
```

のような節の集合で定義してあると考えられるので，モード制約としては

$$m/\langle \mathtt{new\_functor}, 1\rangle\langle \mathtt{f}, i\rangle = IN, \quad 1 \leq i \leq 4$$

のかわりに

$$m(\langle \mathtt{new\_functor\_out}, 1\rangle\langle \mathtt{f}, i\rangle) = out, \quad 1 \leq i \leq 4$$

が課され，各引数のトップレベルにしか制約がかからなくなる．そこで，

```
..., new_functor_out(T,f,10), setarg(1,T,E,NE,NT), ...
```

のようにしてsetargで引数値の変更をするとき，新しく与える項NEにも，トップレベル以外にはモード制約がかからなくなる．

ときには，各引数のトップレベルが入力モードの複合項を作成して返したいこともあるかもしれない．これは，作成した複合項を，送信用ストリームの配列として使いたい場合などに起きる．言語仕様の対称性を考慮すると，このような場合のために，要素のモード制約がnew_functor_outとは逆の組込述語を用意しておくのが適当であろう．述語の名前をnew_functor_inとし，これが

$$m(\langle \mathtt{new\_functor\_in}, 1\rangle\langle \mathtt{f}, i\rangle) = in, \quad 1 \leq i \leq 4$$

よりも強い制約をもたないようにするには，たとえば

```
new_functor_in(T,f,4) :- true |
    T=f(X,Y,Z,W), close(X), close(Y), close(Z), close(W)
```

のような節の集合と，節

```
close([]) :- true | true
```

とが用意してあると考えればよい．しかし，new_functor_out だけしか用意しなくても，任意の部分モードをもつデータ t を複合項から出し入れすることは不可能でない．データ t 自身を出し入れするのでなく，t を適当な1引数関数記号 (h とする) で包んだ項 h(t) を出し入れするようにすれば，トップレベルのモード値に関する new_functor_out の制約を満たすことができるからである．

1.9.6 ベクタやストリングとその操作

ベクタおよびストリングは，実用上きわめて重要な KL1 の機能である．ストリングはベクタの特殊な場合と考えることができるので，以下ではベクタについて考察する．

旧来の KL1 では，ベクタ {f,a,b} と複合項 f(a,b) とを同一視することになっていた．同じ項に対して，ベクタとしてアクセスしたり複合項としてアクセスしたりするプログラムの解析も不可能ではないが，パスの概念に基づくモード体系の柔軟さが十分活かせないという問題があった．しかし，両者を区別するならば，ベクタの存在は，モード解析には何ら支障をきたさない．

ベクタは，モード解析の上では，特別な関数記号をもつ複合項と考えるのが適切である．この関数記号をかりに '$VEC' としよう．すると，ベクタを生成する述語 new_vector は，

```
new_vector(V,L) :- true | new_functor_out(V,'$VEC',L)
```

と定義してあると考えることができる．

ベクタの要素にアクセスする述語としては，vector_element と，2種類の set_vector_element が用意されている．これらはそれぞれ，

```
vector_element(V,K,E) :- true | K1:=K+1, arg(K1,V,E)
set_vector_element(V,K,NE,NV) :- true |
    K1:=K+1, setarg(K1,V,NE,NV)
set_vector_element(V,K,E,NE,NV) :- true |
    K1:=K+1, setarg(K1,V,E,NE,NV)
```

と定義してあると考えることができる．三つの中で基本となるのは，5引数の set_vector_element(V,K,E,NE,NV) であり，他は基底項を要素とするベクタのための簡略版と見なせる．

以上の組込述語は節のボディでのみ呼出し可能なものであったが，ガードでは，項 t がベクタかどうかを判定する vector(t)，ベクタかどうかを判定した上でその長さを l に返す vector(t,l) が用意してある．このうち前者はテスト述語であるからガードで呼び出すのが適当であるが，後者の機能は functor に対応するもので，ボディで使用するのを原則とすべきである．

上の考察では，ベクタを，特別な関数記号 '$VEC' をもつ複合項と考えた．しかし現在の KL1 処理系では，ベクタの「主関数記号」を functor で求めると，そのベクタ自身が

返ってくる．逆に，ベクタを「主関数記号」としてもつ項を `new_functor` で作ることはできない．この仕様は，モード解析には別段支障がないものの，再考の余地があろう．

KLIC 処理系では，ベクタを汎用オブジェクト (generic object) 機能を使って実装している [5]．汎用オブジェクト機能は，言語機能の拡張に有効であるとともに，他言語インタフェースとしても使えるが，他言語で記述した機能の仕様が，KL1 の枠組で説明でき，それに well-moding を与えることができるならば，モード解析に支障はない．

なお，ベクタに対する基本操作としては，要素の取出しや変更のほかに，ベクタの分割が考えられる．たとえば `vector_split(`v`,`k`,`v_1`,`v_2`)` という操作を用意し，ベクタ v の第 k 要素の前までを v_1 に，第 k 要素以降を v_2 に返すのである．また分割の逆操作であるベクタの連結操作も必要となろう．これらの機能は，5 引数 `set_vector_element` と同様，不当なモード制約を課することなく提供でき，モード解析の観点からは何ら問題ない．ベクタの中ほどから部分ベクタを切り出す操作や，切り出した部分を他のベクタで置き換える操作も用意してもよいが，ベクタの分割，連結操作を組み合わせてもできる．

ベクタに関する今後の検討課題としては，多次元ベクタがある．KL1 では，2 次元のベクタを，ベクタのベクタとしてプログラムしている．しかし，配列を多用するプログラムでは，多次元ベクタをサポートしたほうがプログラミングの点でも効率の点でも有利である．多次元配列に対する基本演算の設計，その効率的実装，およびモード解析は，興味深く，実用上も重要な検討課題である．

1.9.7 荘園

KL1 では，プログラムの実行の観測と制御のために，荘園機能 [42] を提供している．その機能の詳細は，PIM 上の KL1 処理系と KLIC では異なってくると予想されるが，最も基本となるのはある項 g をゴールと見なして実行する機能，つまり Prolog の `call(`g`)` に対応する機能である．

この `call` については，モード解析上の問題はない．モード解析では，プログラムで扱うデータ構造のすべてのレベルに対し，完全に情報の流れを把握するので，

$$m/\langle \mathtt{call}, 1\rangle\langle f, i\rangle = m/\langle \mathrm{fp}(f), i\rangle, \quad f \in \mathit{Fun},\ i \in N_f$$

と見なせばよいのである．ただしここで fp は，`call` が使っている *Fun* から *Pred* への単射である．

荘園のもう一つ重要な機能として，例外処理機能がある．PIM 上の KL1 は，強力な例外処理機能をもっており，例えば例外を起こしたゴールの代わりに任意のゴールを与え，そのまま荘園の実行を継続することができる．しかし，あまり強力な例外処理機能は，モード解析をはじめとするプログラム解析や，それに基づく最適化を困難にする．プログラム解析や最適化の可能性を (ほとんど) 保つように，例外処理機能を設計するのがよいであろう．

1.9.8 入出力

KL1の入出力の基本はストリーム入出力である．モード解析は，多様なメッセージを扱うストリームにも対処しており，文字や行単位の入出力，基底項の出力，およびバッファの書出し等のための制御メッセージについては，これまでに述べた枠組で明らかに解析可能である．

検討しなければならないのは，変数を含むかもしれない一般の項の入力である．一般の項を入力するためのメッセージとして，PIM上のKL1処理系では，メッセージ **getwt** (入力項を，"wrap された"，つまり構文解析で得られた諸情報を各記号に付加した形式で返す) と，**gett** (入力項を通常の項の形で返す．Prologの **read** に対応) を提供していた[19]．このうち **getwt** は，返ってくるものが基底項であるから問題ないが，**gett** は基底とは限らない項を返す．メッセージ **gett** に対してwell-modingを与えるとすると，このメッセージが出現しうる入力プロセス中のパス p について，$m/p\langle \mathtt{gett}, 1\rangle = OUT$ としなければならない．**gett** は非基底項のみならず，あらゆる基底項を返しうるからである．しかし，$m/p\langle \mathtt{gett}, 1\rangle = OUT$ であるとすると，定理2から，入力項の中の変数の値は **gett** メッセージの受信側が供給するようになっていなければならない．このため，単に入力項を返す1引数の **gett** には，well-modingを与えることができない．

ところがPIM上のKL1には2引数の **gett** があり，入力項の作成に使った変数のプール (データを蓄えたプロセス) へのストリームを，入力項とともに返すことができる．この2引数の **gett** を使えば，入力項の作成に使った変数にプール経由でアクセスし，それらを具体化することにより，最終的には入力項を基底項に具体化することができる．したがって，モード解析の観点からは，この2引数の **gett** を基本とすべきであるといえる．

ただし，変数プールがサポートするメッセージプロトコルとして，プール中のすべての変数を取り出す機能が必要である．これ以外のプロトコルとして，特定の名前の変数を取り出す機能はサポートしてもよいが，アクセスした変数をプールに残すようなプロトコルがあると，強すぎるモード制約がかかってしまう．アクセスした変数は必ず取り出してしまうようなプロトコルに限定すべきであろう．

getwt や **gett** は，ユーザプログラム中に現れない関数記号をもつ項を返すこともありうる．しかしその関数記号は，第1.3節の定義から集合 *Fun* の要素であり，これまでに述べたモード体系で説明のつくものである．またその関数記号は，ユーザプログラムには現れなくても，入力文字列から項を作成する述語の仮想的定義の中には現れているはずである．

なおKL1は，プログラムのデバッグのために副作用に基づく入出力も提供しているが，入力データを基底項に限定しておけば，これらの存在がモード解析の支障になることはない．基底項を入力する述語の引数を OUT，出力する述語の引数を IN と考えればよいのである．

1.10 考察

以上で提案したモード解析手法は，不動点計算による抽象解釈に基づく大域的データフロー解析でなく，個々のプログラム節が局所的に課するモード制約に基づいている．こ

れは，大きなプログラムの分割コンパイルと本質的に相性がよいことを意味している．ただ分割コンパイルの場合，局所的にモードが決定できない単一化のコードは，リンク時に与える必要がある．もう一つの方法として，より多くの目的コードがコンパイル時に決定できるように，大域的な述語のモードを**宣言**してもらうことも考えられよう．この場合は，リンク時のモード解析は，宣言した制約の無矛盾性の検査として機能する．このように，制約に基づくモード体系は，モード宣言，モード検査，モード推論のための統一的な枠組を提供する．

ここで述べたモード体系は，PARLOG [8] のモード体系とはまったく異なる．PARLOG のモードは，基本的には，Kernel PARLOG へのコンパイル時に出力単一化をボディに移すためのものであり，ゴールのある引数が入力と宣言してあっても，そのゴールがその引数の主関数記号を決定しないとは限らない．たとえばプログラム

```
mode p(?).                    ('?' は入力を表わす)
p(X) :- true : X=5.
```

は正しい PARLOG プログラムであり，GHC プログラム

```
p(X) :- true | X=5
```

に対応する．

DEC-10 Prolog [3] のモードもまた，まったく別物であり，ゴール呼出し時点の引数の具体化状態を宣言するものである．一方，我々のモード体系は基本的に，時刻に依存しない性質を扱っている．とは言っても，ゴールの出力パスの値が呼出し時点で具体化していないということを保証してもくれるが．

モード解析は，プログラムの最適化に有用な情報をもたらす．特に，単一化ボディゴールの分散実装を大幅に簡素化することができる．文献 [44, 45, 46] で提案しているメッセージ指向処理方式は，単一化をメッセージ送信にコンパイルするもので，モード情報があって初めて可能となる最適化技法である．モード解析のもう一つ重要な側面は，ネイティブ・コードの生成をより現実的にしてくれることである．(並行) 論理型言語では，実行時に起きるさまざまな例外的状況に対処しなければならないため，ネイティブ・コードの利用はあまり現実的でないと考えられてきた．

モード解析は，デバッグのためにも有効である．GHC プログラミングでは，たった 1 回しかチャネル出現しない変数があると，その値をガードで検査していない限り，プログラムが誤っている可能性が非常に高い．これは例えば，プログラマが変数名を書き誤ったときに起きる:

```
p(X0, ...) :- ... | q(X0,X1), p(X1, ...)   (X0 は X0 の書き誤り)
```

この場合，二つの変数 X0 と X0 は強い制約，つまり $m/\langle \mathrm{p}, 1\rangle = IN$ と $m/\langle \mathrm{q}, 1\rangle = OUT$ を課する．これらは正しい節から得られる制約 $m/\langle \mathrm{p}, 1\rangle = m/\langle \mathrm{q}, 1\rangle$ とは非常に異っており，他の節から得られる制約と矛盾する可能性が非常に高い．

また，プログラマはよく，使い終ったストリームを閉じ忘れる:

```
p([],     Ys) :- true | true   (ボディは Ys=[] の書き誤り)
p([X|Xs1],Ys) :- ... | Ys=[...|Ys1], p(Xs1,Ys1)
```

これはモードの誤りとして検出される．(誤った) 非再帰節が制約 $m/\langle \mathrm{p}, 2\rangle = IN$ を課するのに対し，再帰節は $m(\langle \mathrm{p}, 2\rangle) = out$ を課するからである．

第 1.8 節の証明は，制約 (HF) や (GV) を使っていない．もしモード解析の目的が，第 1.8 節で証明した基本的性質を保証することだけにあるならば，これらの制約はいらないことになる．しかし，第 1.5 節で議論したように，これらの制約を課することには根拠があり，プログラムの誤りを発見したり三項以上の制約を簡約化したりするのに役に立つと期待できる．

提案したモード体系を，抽象解釈の枠組で理解することも可能であろうが，制約に基づく表現のほうがむしろ単純でわかりやすいと信じる．抽象解釈では，再帰的プログラムの性質を捉えるために不動点の繰返し計算を行なうが，我々は，第 1.3 節の冒頭で述べたプログラミング・スタイルに関する仮定によって，繰返し計算でなく単一化によってモードを計算できるようにしている．

Strand [13] では，変数と値との結合を生成するのに，単一化の代わりに代入プリミティブ $(v := t)$ を用いている．しかし，コンパイル時のモード解析を行なわないと，代入ゴールの左辺が変数であることを実行時に検査しなければならない．我々の枠組では，代入プリミティブは，$m(\langle :=, 1\rangle) = out$ および (それから導けることだが) $m(\langle :=, 2\rangle) = in$ であることを暗黙に宣言した単一化であると考えることができる．

Doc [14]，$\mathcal{A}'\mathcal{UM}$ [48] および Janus [25] は，各変数の出現を 2 回に制限し，プログラマの与えるアノテーションによって入力出現と出力出現とを区別することによって，実装を簡素化しようとしている．このアノテーションも，モード宣言と見なすことができるが，その無矛盾性はコンパイル時か実行時に検査しなければならない．アノテーションは，プログラムを読み易くしたりコンパイルを容易にしたりするかもしれないが，基本的には推論可能な情報であり，その完全な付加をプログラマに強制する必然性はない．

第2章　モード体系の実装と経験

Strong moding is turning out to play fundamental roles in concurrent logic programming (or in general, concurrent constraint programming) as strong typing does but in different respects. "Principal modes" can most naturally be represented as feature graphs and can be formed by unification. We built a mode analyzer, implementing mode graphs and their operations by means of concurrent processes and streams (rather than records and pointers). This is a non-trivial programming experience with complicated process structures and has provided us with several insights into the relationship between programming with dynamic data structures and programming with dynamic process structures. The mode analyzer was then applied to the analyzer itself to study the characteristics of the mode constraints it imposed and of the form of large mode graphs. Finally, we show how our framework based on principal moding can be extended to deal with (1) random-access data structures, (2) mode polymorphism, (3) higher-order constructs, and (4) various non-Herbrand constraint systems.

2.1 Strong Moding in Concurrent Logic/Constraint Programming

Historically, two different notions of modes have been studied in logic programming. Modes of the first kind are concerned with reasoning about temporal properties (i.e., time-of-call/exit instantiation states) of variables, and are used for answering questions such as "Is `X` unbound when `p(X)` is called?" They are usually analyzed using abstract interpretation and necessarily depends on "computation rules." Modes of the second kind, which are less well-known and we are going to deal with here, are for reasoning about non-temporal properties of variables, and are intended to answer questions such as "Which occurrence of `X` in the configuration `p(X),q(X),r(X)` may instantiate `X` eventually?" They are independent of how goals are executed and thus can be regarded as a language construct.

Variables in logic programming languages can be viewed as communication channels. A variable may in general have many writers and many readers (blackboard communication), but in most cases variables are used for cooperative communication, namely point-to-point communication (one writer, one reader) or multicasting/broadcasting (one writer, many readers). In both logic programming and concurrent logic programming, it seems important to be able to distinguish between competitive and cooperative

communication and, for the latter case, to infer the communication protocols used.

From our experience, we are strongly confident that variables in concurrent logic languages can be restricted to cooperative communication without loss of expressive power. Rather, by doing so, we will benefit very much from strong moding, as we do from strong typing in many other languages:

- It helps programmers understand their programs better.
- It detects a certain kind of program errors at compile-time.
- It establishes some fundamental properties statically:
  - Well-moded programs do not collapse due to unification failure (failure of unification body goals).
  - All variables are guaranteed to become *ground* terms upon termination.
  - It distinguishes between data with a single reference and those with multiple references. This provides us with basic information for compile-time garbage collection.
- It provides basic information for program optimization:
  - elimination of various runtime checks,
  - (much) simpler distributed unification,
  - message-oriented implementation [46][45].
- It encourages modular programming by making programmers better aware of module interface.

Since the mode system for Flat GHC was proposed [44][46], some attempts have been made to implement the system [30], but no attempts have been made to implement a mode analyzer based on the unification of mode graphs as it appeared in [44]. In this chapter, we describe our first experiences with the graph-based mode analyzer. Also, we show how our framework based on principal moding can be extended to deal with (1) random-access data structures, (2) mode polymorphism, (3) higher-order constructs, and (4) various non-Herbrand constraint systems.

We assume familiarity with the basic idea of the mode system, though it will be described briefly in Section 2.2. Full detail of the mode system, with proofs of the fundamental properties, can be found in [46]. Implications of the mode system are discussed in [36], which contains informal introduction to concurrent logic programming and the mode system as well.

2.2 The Mode System

As discussed in [36], a logical variable with exactly two occurrences can be compared to a signal cable which has a certain structure (e.g., array of wires) and conveys information under some established protocol. A piece of information flowing into the n-th pin of the plug at one end of a cable will come out from the n-th pin at the other end of the cable, which means that two ends/occurrences of a cable/variable should have exactly inverse (i.e., complementary) polarity structures.

A variable with three or more occurrences in a run-time configuration will be used as a *hub* for one-to-many communication. The polarity structures of the terminals of a hub should be given so that for each set of corresponding positions in those structures, exactly one of them is the inlet of information and the others are outlets.

We call variables with exactly two occurrences *linear variables* and other variables *non-linear variables*, where we do not count the second and subsequent occurrences of a variable in a clause head or any of the occurrences in guard goals. We call clauses not containing non-linear variables *linear clauses* and clauses containing non-linear variables *non-linear clauses*.

An argument of a goal can be compared to a socket of a device. To be compatible, a plug and a socket should have opposite polarity structures when viewed from outside.

The purpose of our mode system is exactly to assign polarity structures to the arguments of predicates defining the behavior of goals, so that each part of data structures will be determined cooperatively, namely by *exactly one* goal. If the part has more than one writer goal, the communication is competitive and hence not cooperative. If the part has no writer at all, the communication is not cooperative, though not competitive, because the readers will never get a value.

A mode is a function from the set of paths specifying positions in data structures occurring in goals, denoted P_{Atom}, to the set $\{in, out\}$. Paths here are not strings of argument positions; instead they are strings of $\langle$*symbol, argument-position*$\rangle$ pairs in order to be able to specify positions in data structures that are yet to be formed.

Formally, the sets of paths for specifying positions in terms and atomic formulas are defined, respectively, using disjoint union as:

$$P_{Term} = (\sum_{f \in Fun} N_f)^* \ , \quad P_{Atom} = (\sum_{p \in Pred} N_p) \times P_{Term} \ ,$$

where *Fun* and *Pred* are the sets of function and predicate symbols, respectively, and N_f and N_p are the sets of positive integers up to and including the arities of f and p, respectively.

Mode analysis tries to find a mode $m : P_{Atom} \to \{in, out\}$ under which every piece of communication will be performed cooperatively. Such a mode is called a *well-moding*. A well-moding is computed by constraint solving. Function symbols in a program/goal clause will impose constraints on the possible polarities of the paths at which they occur. Variable symbols may constrain the polarities not only of the paths at which

(HF) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Fun \Rightarrow m(p) = in)$
(if the symbol at p in h is a function symbol, $m(p) = in$),

(HV) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Var \wedge \exists p' \neq p(\widetilde{h}(p) = \widetilde{h}(p')) \Rightarrow m/p = IN)$
(if the symbol at p in h is a variable occurring elsewhere in h, then $m/p = IN$),

(GV) $\forall p, p' \in P_{Atom}\, \forall a \in G(\widetilde{h}(p) \in Var \wedge \widetilde{h}(p) = \widetilde{a}(p')$
$\Rightarrow \forall q \in P_{Term}(m(p'q) = in \Rightarrow m(pq) = in))$
(if the same variable occurs both at p in h and at p' in G, then
$\forall q \in P_{Term}(m(p'q) = in \Rightarrow m(pq) = in)$)

(BU) $\forall k > 0\, \forall t_1, t_2 \in Term((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle})$
(the two arguments of a unification body goal have complementary submodes)

(BF) $\forall p \in P_{Atom} \forall a \in B(\widetilde{a}(p) \in Fun \Rightarrow m(p) = in)$
(if the symbol at p in a body goal is a function symbol, $m(p) = in$),

(BV) Let $v \in Var$ occur $n\ (\geq 1)$ times in h and B at $p_1, \ldots, p_n$, of which the occurrences in h are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & k = 0; \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & k > 0; \end{cases}$$

where the unary predicate $\mathcal{R}$ over finite *multisets* of submodes represents "co-operative communication" between paths and is defined as

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S(s(q) = out \wedge \forall s' \in S \backslash \{s\}\, (s'(q) = in))$$

Fig. 2.1: Mode constraints imposed by a clause h :- G | B.

they occur but of any positions below those paths. The set of all these constraints syntactically imposed by the symbols or the symbol occurrences in a program does not necessarily define a unique mode because the constraints are usually not strong enough to define one. Instead it defines a 'principal' mode that can best be expressed as a mode graph, as we will see in Section 2.3.

Constraints imposed by a clause h :- G | B, where G and B are multisets of atomic formulae, are summarized in Figure 2.1. Here, *Var* denotes the set of variable symbols, and $\widetilde{a}(p)$ denotes a symbol occurring at p in an atomic formula a. A *submode* of m at p, denoted m/p, is a function (from P_{Term} to $\{in, out\}$) such that $(m/p)(q) = m(pq)$. *IN* and *OUT* are submodes that always return *in* or *out*, respectively. An overline, "$\overline{\ }$", inverts the polarity of a mode, a submode, or a mode value.

Unification body goals, dealt with by Constraint (BU), are *polymorphic* in the sense that different goals are allowed to have different modes. To deal with polymorphism, we

give each unification body goal a unique number. General treatment of polymorphism will be discussed in Section 2.5.2.

As an example, consider a `merge` program:

```
merge([],Y,Z) :- true | Z =1 Y.
merge(X,[],Z) :- true | Z =2 X.
merge([A|X],Y,Z0) :- true | Z0 =3 [A|Z], merge(X,Y,Z).
merge(X,[A|Y],Z0) :- true | Z0 =4 [A|Z], merge(X,Y,Z).
```

From the third clause, for instance, we obtain the following eight constraints, where "." stands for the constructor of non-empty lists:

$$
\begin{array}{rcll}
m(\langle \texttt{merge}, 1\rangle) & = & in & \text{by (HF) applied to "."} \\
m/\langle =_3, 1\rangle & = & \overline{m/\langle =_3, 2\rangle} & \text{by (BU) applied to } =_3 \\
m(\langle =_3, 2\rangle) & = & in & \text{by (BF) applied to "."} \\
m/\langle \texttt{merge}, 1\rangle\langle ., 1\rangle & = & m/\langle =_3, 2\rangle\langle ., 1\rangle & \text{by (BV) applied to A} \\
m/\langle \texttt{merge}, 1\rangle\langle ., 2\rangle & = & m/\langle \texttt{merge}, 1\rangle & \text{by (BV) applied to X} \\
m/\langle \texttt{merge}, 2\rangle & = & m/\langle \texttt{merge}, 2\rangle & \text{by (BV) applied to Y} \\
m/\langle \texttt{merge}, 3\rangle & = & m/\langle =_3, 1\rangle & \text{by (BV) applied to Z0} \\
m/\langle =_3, 2\rangle\langle ., 2\rangle & = & \overline{m/\langle \texttt{merge}, 3\rangle} & \text{by (BV) applied to Z}
\end{array}
$$

From the entire set of clauses, we obtain 24 constraints, of which 6 are of the form $m(p) = in$, 12 are of the form $m/p_1 = m/p_2$, and 6 are of the form $m/p_1 = \overline{m/p_2}$. Elimination of the constraints on $=_k$, however, leaves only four constraints:

$$
\begin{aligned}
m(\langle \texttt{merge}, 1\rangle) &= in \\
m/\langle \texttt{merge}, 1\rangle\langle ., 2\rangle &= m/\langle \texttt{merge}, 1\rangle \\
m/\langle \texttt{merge}, 2\rangle &= m/\langle \texttt{merge}, 1\rangle \\
m/\langle \texttt{merge}, 3\rangle &= \overline{m/\langle \texttt{merge}, 1\rangle}
\end{aligned}
$$

We could handle these constraints as logical formulae, but mode graphs described below allow us to represent and manipulate constraints efficiently.

2.3 Mode Graphs and Principal Modes

It turns out that most of the mode constraints are either of the six forms: (i) $m(p) = in$, (ii) $m(p) = out$, (iii) $m/p = IN$, (iv) $m/p = OUT$, (v) $m/p_1 = m/p_2$, or (vi) $m/p_1 = \overline{m/p_2}$. We call (i)–(iv) *unary* constraints and (v)–(vi) *binary* constraints.

A set of binary and unary mode constraints can be represented as a feature graph (feature structures with cycles), called a *mode graph*, in which

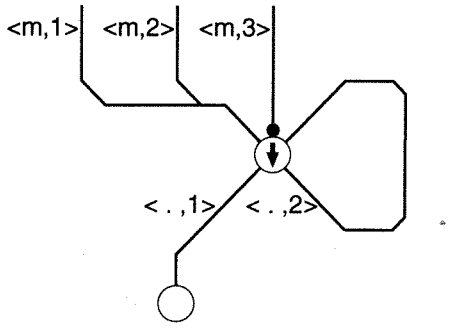

Fig. 2.2: Mode graph of the `merge` program.

1. paths represent paths in P_{Atom},
2. nodes may have mode values determined by unary constraints,
3. arcs may have "negative signs" that invert the interpretation of the mode values beyond those arcs, and
4. binary constraints are represented by the sharing of nodes.

Figure 2.2 is the mode graph of the four constraints from the `merge` program.

An arc of a mode graph represents the pair of a predicate/function symbol (abbreviated to its initial in the figures) and an argument position. The pair exactly corresponds to a feature of a feature graph. A sequence of features forms a path both in the sense of our mode system and in the graph-theoretic sense.

A node is possibly labeled with a mode value (*in* shown as "↓", or *out* shown as "↑") to which any paths p_1, p_2, ... terminating with that node are constrained, or with a constant submode (*IN* shown as "↓" with a grounding sign (as in Figure 2.4), or *OUT*) to which the submodes m/p_1, m/p_2, ... are constrained.

An arc is either a negative arc (bulleted in the figures) or a positive arc. When a path passes an odd number of negative arcs, that path is said to be *inverted*, and the mode value of the path should be understood to be inverted. Thus the number of bulleted arcs on a path determines the *polarity* of the path.

A binary constraint of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$ is represented by a shared node with two (or more) incoming paths with possibly different polarities. When the polarities of the two incoming paths are different, the shared node stands for complementary submodes; otherwise the node stands for identical submodes.

Figure 2.2 has a node, under the arc labeled $\langle ., 1 \rangle$, that expresses no information at all. It was created to express binary constraints, but all its parent nodes were later merged into a single node by other constraints.

As another example, consider a program that simply unifies the two arguments:

```
p(X,Y) :- true | X=Y.
```

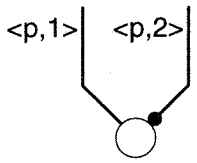

Fig. 2.3: Mode graph of the unify program.

The program forms a simple mode graph shown in Figure 2.3. This graph can be viewed as the *principal mode* of the predicate `p`, which represents many possible particular modes satisfying the constraint $m/\langle \mathtt{p}, 1\rangle = \overline{m/\langle \mathtt{p}, 2\rangle}$. In general, the principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary or binary.

Constraints imposed by the rule (BV) may be non-binary. Non-binary constraints are imposed by non-linear variables, and cannot be represented as mode graphs by themselves. However, by *delaying* them, most of them will be reduced to unary/binary ones by other constraints, as we will see later. In this case they can be represented in mode graphs, and the programs that imposed them have unique principal modes (as long as they are well-moded).

Theoretically, some non-binary constraints may remain unreduced, whose satisfiability must be checked eventually. However, a much more practical solution is to let programmers declare the modes of the paths where non-linear variables occur.

The union (i.e., conjunction) of two sets of constraints can be computed efficiently as unification over feature graphs. For instance, adding a new constraint $m/p_1 = m/p_2$ causes the subgraph rooted at p_1 and the subgraph rooted at p_2 to be unified. A good news is that an efficient unification algorithm for feature graphs has been established [1].

Figure 2.4 shows the mode graph of a quicksort program using difference lists. The head and the tail of a difference list, namely the second and the third arguments of `qsort`, are constrained to have complementary submodes.

Figure 2.5 shows the driver of a demand-driven sequence generator that receives messages `done` or `more` from an I/O stream and keeps sending requests to the sequence generator until `done` is received. The figure shows how mutual recursion can be dealt with: `driver` calls `checkinput` after sending a message and `checkinput` calls `driver` after sending two messages. These two predicates form a cycle with three nodes in the mode graph.

2.4 Implementing Mode Analysis

We have implemented a mode analyzer for Flat GHC, and have extended it to deal with most features of KL1 [42]. The analyzer is itself a well-moded program entirely written in KL1. Mode analysis proceeds as follows:

```
qsort([],     Ys0,Ys ) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
    part(X,Xs,S,L),qsort(S,Ys0,[X|Ys2]),qsort(L,Ys2,Ys3).
part(_,[],     S, L ) :- true | S=[],L=[].
part(A,[X|Xs],S0,L ) :- A>=X | S0=[X|S],part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0=[X|L],part(A,Xs,S,L).
```

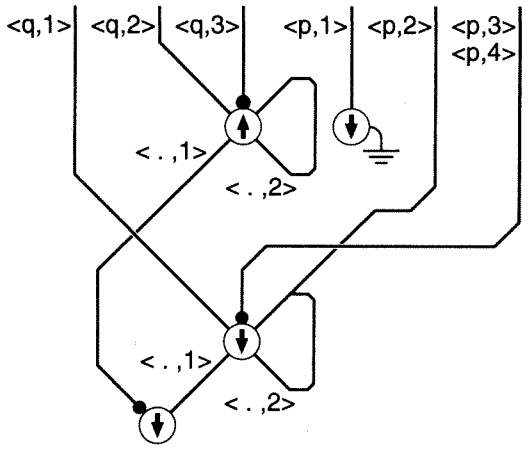

Fig. 2.4: A quicksort program and its mode graph.

```
driver(Fs,IOs0) :- true | IOs0=[gett(X)|IOs1],checkinput(Fs,IOs1,X).
checkinput(Fs, IOs, done) :- true | Fs=[],IOs=[].
checkinput(Fs0,IOs0,more) :- true |
    Fs0=[N|Fs1],IOs0=[putt(N),nl|IOs1],driver(Fs1,IOs1).
```

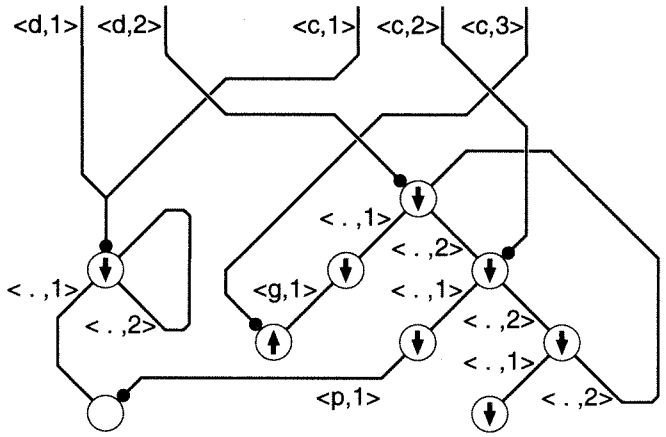

Fig. 2.5: A mutually recursive program and its mode graph.

1. Constraint generation
   (a) KL1 clauses are translated into their normal forms [46].
   (b) Calls to polymorphic built-in predicates are identified and numbered.
   (c) A symbol table is generated, which records all the occurrences of predicate, function, constant, and variable symbols.
   (d) Constraints imposed by symbols, occurrences of symbols, and built-in predicates are generated according to the rules in Figure 2.1.
2. Constraint solving
   (a) The constraints are put into an empty mode graph to form a graph representing their conjunction.
   (b) If successful, the final graph state is retrieved; otherwise, failure and its reason are reported.

2.4.1 Constraint Generation

Since the first part, constraint generation, is simply the syntactic manipulation of a given program, we only note that KL1 differs from Flat GHC in the following aspects, which are all handled appropriately.

1. KL1 supports modularization; a predicate is identified by the pair of the module it belongs to and the predicate name.
2. A program commences execution by calling the nullary predicate `main` in the module `main`.
3. KL1 supports vectors. A vector is denoted $\{element_1, \ldots, element_n\}$, which is regarded as a structure with a special function symbol indicating that the term is a vector of length n.
4. KL1's guard built-in predicates may have output arguments (e.g., `functor`), which is not allowed in Flat GHC. They are treated as a source of information like input arguments in the head. Output arguments in a clause guard may be used both for providing values to be used in the body (cooperative communication) or for checking the applicability of the clause (competitive communication).
5. KL1 features character strings and string operations, but a string can be treated as a constant because it cannot contain uninstantiated variables.

2.4.2 Constraint Satisfaction

Our constraint solver represents mode graphs using processes and streams. Although not shown in the figures, each mode graph has a root node corresponding to the empty path.[1] New constraints are added by sending messages to the root node. Retrieval of

[1] P_{Atom} does not contain an empty path ϵ, but we could extend the domain of modes to $P_{Atom} \cup \{\epsilon\}$ and let $m(\epsilon) = in$.

mode information can be done by sending messages as well.

Process representation is interesting in its own right as an experiment of programming complicated process structures. We must deal with (1) graphs with node sharing and cycles and (2) the merging of graph nodes. It is not clear whether they can be programmed in a strongly moded framework and how much parallelism can be exploited.

One alternative to the process representation would be to use rational trees for representing mode graphs. Unfortunately, even if the underlying language features rational trees, we cannot rely too much on the built-in unification but must define unification for feature graphs ourselves. One reason is that the unification procedure should be able to report failure explicitly. Another obvious alternative would be to represent mode graphs using arrays of nodes and arcs, modifying them as new constraints are added. The sequential nature of this approach can make global operations such as termination detection easier to implement. However, we chose the process approach to explore the viability of process representation of dynamic data structures and to be able to exploit parallelism in future (Section 2.4.4).

The `node` process representing an ordinary node contains the following arguments:

1. an input stream for receiving messages,
2. a node identifier used for the equality checking of nodes,
3. a mode value (*unconstrained*, *input*, or *output*),
4. a list of features corresponding to output streams,
5. a list of output streams,
6. a termination variable shared by all node processes, and
7. a flag for the retrieval of the graph state.

When there is more than one incoming arc (stream) to a node, they are merged using frontend `merge` processes. A negative arc is represented using an inverter filter process that inverts the interpretation of mode values contained in messages.

A node is created by sending a message to the node server, which provides each node with its identifier. Unlike in procedural languages, the identity of nodes cannot be checked by pointer comparison but this capability should be provided explicitly by the process. Assignment (pointer copying) and equality checking (pointer comparison) are provided as basic operations in many programming languages, but there are cases where they should not be allowed.

A node whose submode is constrained to *IN* or *OUT* is represented by a `gnode` process, which contains four of the above seven arguments: 1, 2, 3, and 7.

Operating on Mode Graphs

All operations on mode graphs are provided as messages to the root node, which are delegated to appropriate processes. The delegation of messages corresponds to the dereferencing of pointers in procedural languages.

Operations (i.e., messages) accepted by a node process include the following:

1. examine the state of the node,
2. instantiate the mode value of the node,
3. add a new outgoing arc to the node,
4. create and return a new input stream to the node,
5. unify the node with another node,
6. examine the state of the (sub)graph rooted at the node, and
7. forward the above requests to an offspring node.

Operation 4 corresponds to pointer assignment in procedural programming.

Unification of two nodes is a 'symmetric' operation that the object-oriented programming style is not very good at. The operation is divided into two phases: It first accesses one of the nodes to obtain an input stream to it, and then sends a `unify_with` message to the other node. Care must be taken so that the second phase does not start until the first phase reaches the target node. Otherwise the second phase may reach its target node earlier, blocking the first phase to be delegated to its target node. This was actually the most awkward error we first made in implementing the constraint solver.

Thanks to the monotonicity of the constraint framework, however, the first and the second phases of the unification operation can be intervened by another operation.

The `unify_with` message takes the input stream to, and the identifier of, the partner node to its own target node. If the two nodes turn out to be the same, unification simply succeeds or fails depending on whether the polarities of the two paths are the same or not. Otherwise it examines the state of the partner node. If the two nodes have compatible mode values, they are unified by (1) merging the input streams of the nodes and directing the result to one of the nodes, (2) terminating the other node, and (3) merging the two sets of outgoing arcs, unifying corresponding arcs from each set recursively.

Retrieval of Graph States

Retrieval of the state of the whole graph is not straightforward due to the circularity. We have prepared two messages for the purpose: `examine` and `examined`. The `examine` message is propagated over the graph, splitting itself at nodes with two or more outgoing arcs and turning the flag of each node on, until it reaches nodes with the flags on, and collects the states of the nodes using difference lists. The `examined` message is simply for turning off all the flags. We could dispense with the `examined` message by employing toggling flags rather than set-reset flags.

For a snapshot of the dynamically changing cyclic graph to be meaningful, no messages that may alter the graph should be issued before the processing of `examine` terminates. This will not cause a performance problem because snapshots will not be taken so often.

Termination

Termination of a circular process structure turned out to be not so straightforward. Stream closing corresponds to the removal of a pointer in procedural programming. However, because of the circularity, propagating the stream closing operation from the root node to child processes is not enough to close all the streams and thus terminate all the node processes. So we decided to have all the node processes share a termination variable, which will be instantiated to `abort` when the graph is to be terminated.

However, each node process cannot simply terminate itself when it finds that the termination variable has been instantiated. Since the input stream of a node has a sophisticated protocol that may require backward communication, the node cannot discard it freely but must wait until it is closed: This is an example of the '*data-as-resource*' moral enforced by strong moding. Upon instantiation of the termination variable, each process closes its own output streams, waits until its input stream is closed, and only after that it terminates gracefully.

2.4.3 An Experiment—Analyzing the Mode Analyzer

As an experiment, we analyzed the constraint solver of our mode system, which had 190 clauses.

Those 190 clauses imposed 2464 constraints in total, which were classified as Table 2.1 according to the forms of the constraints and the rules that imposed the constraints. "Built-in" stands for the constraints imposed by calls to built-in predicates.

The most remarkable thing about these statistics is that, of 1392 constraints imposed by Constraint (BV), more than 90% were of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$. Thus we can say that the clauses analyzed are highly linear. Only 5% of the variables were singletons, and 3% had more than two occurrences and imposed non-binary constraints. 2% of the variables had their values examined in guards, for which Constraint (BV) were weakened to a form $m(p) = in$ [46].

All of the 42 non-binary constraints were reduced to unary or binary constraints using other unary or binary constraints. Actually they were reduced to 6 constraints of the form $m/p_1 = \overline{m/p_2}$ and 72 constraints of the form $m/p = IN$. This means that non-linear variables were all used under simple, unidirectional communication protocols.

The final mode graph contained 162 nodes and 938 arcs. Table 2.2 shows the depth of the nodes from the top node. Considering that the program analyzed used quite complicated protocols, this result suggests that mode graphs are, in general, very shallow and wide. The program used complicated protocols (e.g., streams of messages containing other streams), but it defined various local procedures that had access to and handled various parts of the protocols. Mode graphs obtained by larger programs will be wider due to many top-level features corresponding to predicate arguments, but they will not be too deeper.

Of the 938 arcs, 814 went from the top-level node and were labelled with predicate

Table 2.1: Constraints imposed by the mode constraint solver.

Type	Rule	Number of constraints
$m(p) = in$	(BF)	453
	(HF)	288
	(GV)	54
	(BV)	24
	Built-in	22
$m(p) = out$	Built-in	18
$m/p = IN$	(BV)	69
	(GV)	2
	(HV)	4
$m/p_1 = m/p_2$	(BV)	1074
$m/p_1 = \overline{m/p_2}$	(BV)	183
	(BU)	231
Non-binary	(BV)	42

Table 2.2: Depth of the nodes of the mode graph of the mode constraint solver.

Level	Number of nodes
0	1
1	124
2	22
3	15
> 3	0

arguments, while 124 went from non-top-level nodes and were labelled with function arguments. Of the 814 arcs sharing 124 level-1 nodes, 462 were by polymorphic unification body goals and 36 were by arithmetic goals which were also polymorphic. The remaining 316 arcs were those corresponding to the arguments of user-defined predicates.

Of the 161 non-top-level nodes, 122 had no outgoing arcs and the remaining 39 nodes had the total of 124 outgoing arcs. The node representing the path of the messages to the input stream of the predicate `node` had 54 outgoing arcs, while the other 38 nodes had less than two arcs on average. This means that hash tables should be used for maintaining the set of outgoing arcs of the top-level node, while simpler data structures can be used for other nodes.

2.4.4 Parallelism

Although mode analysis is not a highly computation-intensive task, it is worthwhile to explore the possibility of parallel speedup. One prominent feature of our mode system is that inter-procedure global analysis is done simply as incremental constraint solving which has much potential for parallel execution.

The first phase of mode analysis, constraint generation, is a highly parallel task because each clause yields its own mode constraints independently.

The second phase, constraint solving, is worth closer look. An important advantage of the constraint framework is that constraints can be merged in any order. Moreover, in our case, the order will not affect the performance too much. Thus the simplest and the most practical way of parallel execution is to exploit coarse-grain parallelism by creating a mode graph for each procedure or each module independently and merging them later.

Fine-grain parallelism that could be obtained by the pipelined processing of messages is subtler. Firstly, as we saw in Section 2.4.3, mode graphs are not deep anyway. Secondly, as we saw in Section 2.4.2, unification of two nodes imposes certain sequentiality. However, it is still important for mode graphs to be able to process messages concurrently, because imposing too much sequentiality between messages leaves less freedom (on the part of implementation) in the scheduling of message handling and can lead to lower sequential performance. Optimizing compilers may well exploit independence of primitive operations to gain performance.

Fortunately, in most cases, a message can be sent to mode graphs before the previous message finishes processing. Because a mode graph becomes "constrained" monotonically, concurrent instantiation of nodes and concurrent unification of nodes will not lead to an incorrect state as long as the atomicity of primitive operations such as the instantiation of node values and the merging of nodes is guaranteed. A message may enter a mode graph even if the previous one causes a mode error. Its effect is simply that when some message causes and reports a mode error, subsequent messages may

Fig. 2.6: The predicate `set_arg` and its mode graph.

already have given additional constraints to the mode graph.

2.5 Extension of the Mode System

2.5.1 Arrays

There have been a number of proposals of mutable array constructs that hide side effects at the language level but exploit them in implementation; some early work include [10], [12], and [41]. Not a few symbolic languages lack array constructs, but they are essential in many real-life applications.

KL1 supports several built-in operations for accessing and updating the elements of vectors and compound terms. In general, the semantics of built-in predicates can be explained by means of a possibly infinite number of virtual clauses, and the principal modes of built-in predicates can be obtained by considering the mode constraints imposed by those virtual clauses.

For array constructs, mode analysis tells us that the most basic element access operation, namely the operation that has the most general principal mode, is

```
set_arg(I, T0, X0, X, T).
```

This operation receives an index value `I` and a (compound) term `T0`, and returns through `X0` the `I`th value of `T0`. In addition, it returns through `T` a compound term which is identical to `T0` except that the `I`th element is replaced by `X`. A similar operation is defined for vectors as well [36]. Figure 2.6 shows the operation and the mode graph of `set_arg`, where $\langle ?, ? \rangle$ stands for a wildcard that matches any feature. Note that different elements of an array are constrained to have identical modes, but the mode of the elements itself is not constrained at all.

Strong moding is deeply concerned with the number of access paths to each variable. As a result, data structures have an aspect of *resources* in general, whose access paths should not be copied or discarded freely. As can be seen in Figure 2.6, an array element should, by default, be removed from the array once accessed, and the resulting blank should be filled with another value.

Array creation is another fundamental operation. In Prolog, `functor` initializes the

arguments of the created structure with distinct fresh variables, which are *instantiated* afterwards if necessary. However, strong moding tells us that the arguments should be initialized with constants and be *updated* by `set_arg`.

Strong moding is particularly important in array processing because it may enable update-in-place. Let the nodes of a mode graph have a *shared/non-shared* flag as well as mode values. *Shared/non-shared* means that the paths ending at the node may/won't be used for one-to-many communication, respectively. To see at which paths shared data may occur, we set the flags of all the nodes at the paths where non-linear variables occur, and of all the nodes below the nodes with the flags on. Then we can see what paths will be used only for one-to-one communication, and arrays occurring at those *non-shared* paths can be updated in place.

Aliasing is recognized as an awkward phenomenon in procedural programming, in which `a[`e_1`]` and `a[`e_2`]` may or may not denote the same variable depending on whether e_1 and e_2 evaluate to the same value. However, accessing two elements of a non-shared array using `set_arg` will not create new, implicit sharing. To access the `Ith` element and the `Jth` element of an array `A`, one will call `set_arg` twice:

```
set_arg(I, A, AI, AInew, A1), set_arg(J, A1, AJ, AJnew, A2).
```

The array `A1` does not contain the original `Ith` element any more, so `AI` and `AJ` cannot be the same unless `AI` and `AInew` happens to be the same. However, `AI` and `AInew` cannot be the same as long as the array elements occur only at *non-shared* positions. For them to be the same, `AI` must occur in a goal for equating it with `AInew` in addition to the occurrence in `set_arg`, but then, we cannot 'use' `AI` through its third occurrence because it does not exist by the '*non-shared*' assumption. `AI` and `AJ` could be the same if one replaced `A1` in the second call by `A`, but then the array itself would become a shared array.

2.5.2 Polymorphic Modes

A unification body goal is polymorphic in the sense that its different occurrences in program text may have different modes as long as they obey Constraint (BU). Constraint (BU) here is considered to represent the 'principal mode scheme' for unification, and different occurrences may have different instances of it.

Array operations, stack processes, and stream merging are examples of generic programs in the sense that they do not constrain the modes of elements. Different arrays, stacks, or stream mergers should be able to accept elements with different protocols, where the necessity of polymorphic modes arises.

Although not yet implemented, polymorphism could be incorporated easily. For polymorphic predicates, their principal mode schemes (i.e., mode graphs) are computed first. To allow different instantiations of a principal mode scheme, a *copy* of the mode graph representing the principal mode scheme will be created for each call to a

polymorphic predicate, which will be merged into the mode graph of the whole program. (In the monomorphic case, the original graph of the predicate is simply merged into the mode graph of the rest of the program.)

It seems that polymorphic predicates should be declared so in some way. Such a distinction was done also when introducing type polymorphism into functional languages. In ML, for instance, $(\lambda x.A)E$ and `let` $x = E$ `in` A have different meanings if types are taken into account.

The above treatment of polymorphism requires that the mode schemes of polymorphic predicates be obtained before analyzing the rest of the program that uses the polymorphic predicates. So polymorphic predicates should be *stratified* so that mode analysis can start from the 'most polymorphic' predicates that depend on no other predicates and end with the analysis of the whole program.

2.5.3 Higher-Order

There are two possible ways to allow a goal to dynamically determine the predicate to be called: One is `call` (analogous to `eval` in Lisp) and the other is `apply`.

Let `call`(G) be a goal that interprets G as a goal (by interpreting the principal function symbol as the predicate to be called) and executes it. The moding of `call` is straightforward; it simply imposes the constraint $m/\langle \texttt{call}, 1\rangle = m$.

In contrast, `apply` needs extension to the mode system. Suppose `apply`(P,X,Y) is a goal that executes a binary predicate P with the arguments X and Y. P may be either a function symbol representing a predicate to be called, a list of clauses (in which bound variables are represented by constants), or a compiled code with mode information. In either case, P is a ground term, but should have a mode as a predicate as well. The mode of `apply` could be represented as the left graph of Figure 2.7. Here, dotted lines represent the constraint that, when the first argument of `apply` is interpreted as a program, the first/second argument of that program must have the identical mode as the second/third argument of `apply`, respectively.

Then consider a predicate that applies P to X twice:

```
twice(P,X,Z) :- apply(P,X,Y), apply(P,Y,Z).
```

If `apply` is monomorphic, applying Constraint (BV) to the variables `X`, `Y`, and `Z` will result in the right graph of Figure 2.7 (where the constraints on `apply` is omitted).

2.5.4 Non-Herbrand Constraint Systems

Concurrent constraint programming generalizes concurrent logic programming by allowing data types that are not based on syntactic equality over the Herbrand universe (set of finite ground terms). Here we consider three extensions.

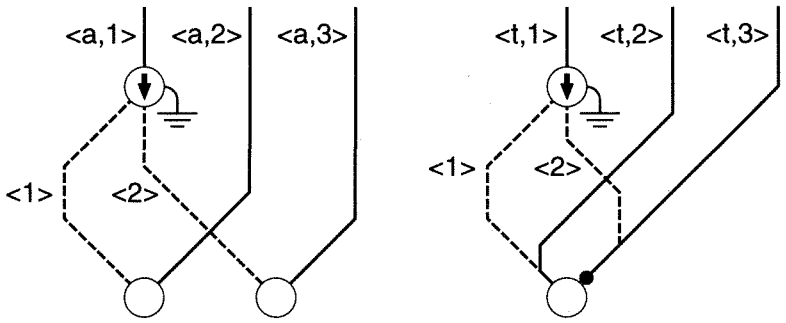

Fig. 2.7: Moding higher-order predicates.

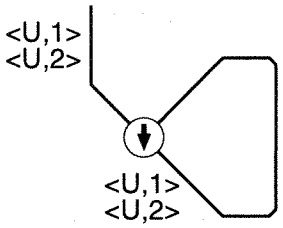

Fig. 2.8: Constraints imposed by an associative and commutative operator $\cup$.

Rational terms: Our path-based mode system can quite naturally deal with non-finite rational terms. Readers may have noticed the similarity between the way our mode is defined and the way infinite trees are represented as functions.

Numerical constraints: Numerical constraints can be dealt with in a moded framework if dataflow can be determined statically. For instance, if the constraint goal `X=Y+1` is used always for determining `Y` from `X` or always for determining `X` from `Y`, it can be moded. However, dataflow caused by solving simultaneous equations will not be that simple in general.

Equational theories: Syntactic equality can be replaced by various equational theories, and a lot of work has been done on unification under equational theories. Here we focus on simple built-in theories; associativity, commutativity and idempotency.

Associativity and commutativity have the property that rewriting based on those preserve the number of occurrences of symbols. Actually they can be included naturally into the mode system. For instance, bags (multisets) enjoy the properties $t_1 \cup t_2 = t_2 \cup t_1$ and $t_1 \cup (t_2 \cup t_3) = (t_1 \cup t_2) \cup t_3$. So the paths where bags may occur should obey the constraint shown in Figure 2.8. That is, any subterm of a bag whose parent symbols are all $\cup$ (the bag constructor) must have an identical mode whose top-level is *in*.

On the other hand, idempotency ($t = t\ op\ t$) says that terms can be freely copied and two identical terms can be freely contracted. This is not very compatible with

the data-as-resource view, and any path which is an inlet/outlet of an idempotent operator will be constrained to IN/OUT, respectively.

2.6 Related Work

Tick and Koshimura implemented and compared several algorithms for mode analysis [30]. One of them uses process structures to represent mode graphs, but their mode graphs have many differences from the graphs in [44] and [46]. They form 'initial mode graphs' and minimize them to form the final graphs, while we add many simple constraints (in the form of messages) to an empty mode graph to form the final graphs. Their graphs deal with non-binary constraints, while we delay non-binary constraints to avoid complication. Their graphs require additional information called 'partition node sets' to maintain the node identifiers, which we need not have. They implement the unification of cyclic structures using marking, while we dispense with marking by implementing it using incremental redirection of streams.

2.7 Conclusions

We implemented a mode analyzer for Flat GHC and KL1, itself described as a strongly moded KL1 program. The analyzer was applied to the analyzer program itself, which used fairly sophisticated communication protocols, to see if automatic mode analysis worked well for non-trivial KL1 programs. We also discussed how the current moding framework could be extended to deal with random-access data structures, polymorphic modes, higher order, and general constraint systems.

Our implementation of the mode system employs quite sophisticated process structures, namely feature graphs with cycles and node sharing. Concurrent operations on such data structures involve nondeterminism and make programs harder to debug. Also, operations on graphs include a rather unusual operation: the merging of two nodes. We succeeded in describing all these in a strongly moded framework and made sure that strongly moded concurrent logic programming was expressive enough for quite complicated programs. Rather, we benefited much from the mode system in debugging.

In spite of complicated process structures formed, debugging was not so difficult. Bugs had to be found manually at first, but most of them were those which could be detected by mode analysis. Many of the bugs we removed later were detected by the mode analyzer itself.

Not all bugs were identified easily, however. The most awkward one was perpetual suspension resulting from the misunderstanding of causality between messages handled concurrently in a nondeterministic program. However, exploiting concurrency is important both for parallel execution and efficient sequential execution.

Self-application of the mode analyzer has confirmed our conjectures that (1) most variables are used for one-to-one communication (i.e., are linear) and (2) non-binary

constraints will be reduced to unary/binary constraints. However, these points require further study because programs written by other people may be less linear.

For the mode system to be more practical, it should generate a user-friendly error messages to non-well-moded programs. The current system simply reports the constraint that finally caused inconsistency and the mode graph immediately before the inconsistency was detected. However, a more user-friendly system should find as concise and intuitive an *explanation* of inconsistency as possible. The next chapter will report an algorithmic approach to the diagnosis of non-well-moded programs.

第3章　モード体系を利用した誤り検出

Strong moding and constraint-based mode analysis are expected to play fundamental roles in debugging concurrent logic/constraint programs as well as in establishing the consistency of communication protocols and in optimization. Mode analysis of Moded Flat GHC is a constraint satisfaction problem with many simple mode constraints, and can be solved efficiently by unification over feature graphs. In practice, however, it is important to be able to analyze non-well-moded programs (programs whose mode constraints are inconsistent) and present plausible "reasons" of inconsistency to the programmers in the absence of mode declarations.

This chapter discusses the application of strong moding to systematic and efficient static program debugging. The basic idea, which turned out to work well at least for small programs, is to find a minimal inconsistent subset from an inconsistent set of mode constraints and indicate the symbol(occurrence)s in the program text that imposed those constraints. A bug can be pinpointed better by finding more than one overlapping minimal subset. These ideas can be readily extended to finding multiple bugs at once. For large programs, stratification of predicates narrows search space and produces more intuitive explanations. Stratification plays a fundamental role in introducing mode polymorphism as well. Our experiments show that the sizes of minimal subsets are small enough for bug location and do not depend on the program size, which means that diagnosis can be done in almost linear time.

3.1 Introduction

One of the prominent features of concurrent logic/constraint programming languages is that they allow us to describe interprocess communication with complicated protocols quite easily. Data structures with complicated dataflow, such as streams of messages with reply boxes and streams of streams, can be expressed without any extensions to their simple, basic computation model.

However, most implementations of concurrent logic languages detect fundamental bugs, such as connecting two streams with different communication protocols, as run-time errors. These should ideally be detected statically. Static type systems, if available, may detect some of those bugs, but do not suffice to ensure the consistency of communication protocols used in a whole program. Thus we need a framework of information flow analysis—a mode system. Mode analysis is very useful for program optimization as well as for the static detection of bugs [46].

Most frameworks of mode analysis proposed so far for concurrent and ordinary logic programming languages were based on abstract interpretation. In contrast, the mode system proposed by one of the authors [46] is constraint-based, that is, mode analysis means to solve a system of mode constraints imposed by individual symbols or symbol occurrences in a program. Since the analyzer does not have to trace execution paths, the framework is particularly useful for the analysis of parallel and concurrent languages in which primitive operations are only partially ordered. Another advantage is that it is inherently amenable to the separate analysis of large programs.

Moded Flat GHC [46] takes those advantages of the constraint-based mode system and incorporates it as a language construct rather than just as a framework for program analysis.[1]

An efficient algorithm of mode analysis has already been established for well-moded programs [46]. However, it was not clear how to find, for a non-well-moded program, a plausible "reason" of mode errors efficiently. Since the principal mode of a program is determined by the conjunction of all mode constraints, non-well-modedness means that the conjunction is inconsistent (that is, there are no modes satisfying all the constraints). However, simply reporting that the conjunction of all the mode constraints is inconsistent does not help debugging large programs. The purpose of this chapter is to propose practical algorithms that locate the reasons of mode errors as precisely and efficiently as possible.

One may wonder how many of the bugs can be detected by rather simple mode analysis, but our experience has shown that that surprisingly many of them can [37]. Bugs which cannot be identified by mode analysis are likely to be related to problems with algorithms.

3.2 Strong Moding and Mode Analysis

This section outlines the mode system of Moded Flat GHC and the associated mode analysis. Due to space limitations, readers unfamiliar with Moded Flat GHC are referred to [36] for introduction and [46, 37] for technical details and proofs of fundamental properties.

The purpose of the mode system of Moded Flat GHC is to assign to each predicate argument a polarity structure that defines the direction of information flow of each part of data structures. The polarity structure is computed by mode analysis so that each part of data structures will be instantiated cooperatively, namely by *exactly one* goal.

A mode in our mode system is a function from the set of *paths* for specifying each "part" of data structures to the two-valued codomain $\{in, out\}$. Paths here are strings of pairs, of the form $\langle symbol, arg \rangle$, of predicate/function symbols and argument posi-

[1] Strong moding can be incorporated in ordinary logic programming languages as well [29], but it is particularly important in concurrent logic programming because most concurrent logic programs have fixed dataflow and make more restricted use of unification. However, the diagnosis technique proposed in this chapter is quite general.

tions. Formally, the set P_{Term} of paths for terms and the set P_{Atom} of paths for atomic formulae are defined using disjoint union as:

$$P_{Term} = (\sum_{f \in Fun} N_f)^*, \; P_{Atom} = (\sum_{p \in Pred} N_p) \times P_{Term} \; ,$$

where $Fun/Atom$ are the sets of function/predicates symbols, and N_f/N_p are the sets of possible argument positions (numbered from 1) for the symbols f/p.

The purpose of the mode system is to find a mode $m : P_{Atom} \rightarrow \{in, out\}$ under which every piece of communication is cooperative. Such a mode is called a *well-moding*. Intuitively, *in* means the inlet of information and *out* means the outlet of information.

Well-modings can be computed by solving mode constraints imposed by (i) the occurrences of function symbols and (ii) the variable symbols in a program and an initial goal clause. A program does not usually define a unique well-moding but has many of them. So the purpose of mode analysis is to compute the set of all well-modings in the form of a *principal* (i.e., most general) mode. Principal modes can be expressed naturally by mode graphs, as described later in this section.

Given a mode m, we define a *submode* m/p, namely m viewed at the path p, as a function satisfying $(m/p)(q) = m(pq)$. We also define IN and OUT as submodes always returning *in* and *out*, respectively. An overline '$-$' inverts the polarity of a mode, a submode, or a mode value.

A Flat GHC program is a set of clauses of the form h :- G | B, where h is an atomic formula and G and B are multisets of atomic formulae. Constraints imposed by a clause h :- G | B are summarized in Figure 3.1, where $\widetilde{a}(p)$ means a symbol occurring at the path p in an atomic formula a, Var is the set of variable symbols, and $Term$ is the set of terms defined over Fun and Var. Rule (BU) numbers unification body goals because the mode system allows different body unification goals to have different modes. This is a special case of mode polymorphism we will discuss in Section 3.6.

As an example, consider the following program for stream merging:

```
m([],Y,Z) :- true | Z =1 Y.
m(X,[],Z) :- true | Z =2 X.
m([A|X],Y,ZO) :- true | ZO =3 [A|Z],m(X,Y,Z).
m(X,[A|Y],ZO) :- true | ZO =4 [A|Z],m(X,Y,Z).
```

The third clause, for example, imposes the following eight constraints (“.” stands for the constructor of non-empty lists):

(HF) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Fun \Rightarrow m(p) = in)$
(If a function symbol occurs at p in h, $m(p) = in$.)

(HV) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Var \wedge \exists p' \neq p(\widetilde{h}(p) = \widetilde{h}(p')) \Rightarrow m/p = IN)$
(If the symbol at p in h is a variable occurring elsewhere in h, $m/p = IN$.)

(GV) $\forall p, p' \in P_{Atom} \forall a \in G(\widetilde{h}(p) \in Var \wedge \widetilde{h}(p) = \widetilde{a}(p') \Rightarrow \forall q \in P_{Term}(m(p'q) = in \Rightarrow m(pq) = in))$
(If the same variable occurs both at p in h and at p' in G, then $m(pq) = in$ if the path $m(p'q)$ is examined in a guard goal.)

(BU) $\forall k > 0 \forall t_1, t_2 \in Term((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle})$
(The two arguments of a unification body goal have opposite submodes.)

(BF) $\forall p \in P_{Atom} \forall a \in B(\widetilde{a}(p) \in Fun \Rightarrow m(p) = in)$
(If a function symbol occurs at p in a body goal, $m(p) = in$.)

(BV) Let v be a variable occurring exactly $n\,(\geq 1)$ times in h and B at $p_1, \ldots, p_n$, of which the occurrences in h are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & \text{if } k = 0; \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & \text{if } k > 0; \end{cases}$$

where the unary predicate $\mathcal{R}$ over finite *multisets* of submodes represents "cooperative communication" between paths and is defined as

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S(s(q) = out \wedge \forall s' \in S \backslash \{s\}\, (s'(q) = in)).$$

Fig. 3.1: Constraints imposed by a clause h :- G | B.

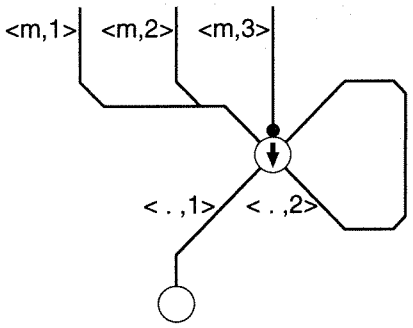

Fig. 3.2: The mode graph of a stream merging program.

$$
\begin{aligned}
m(\langle \mathtt{m}, 1\rangle) &= in && \text{by (HF) applied to ``.''}\\
m/\langle =_3, 1\rangle &= \overline{m/\langle =_3, 2\rangle} && \text{by (BU) applied to } =_3\\
m(\langle =_3, 2\rangle) &= in && \text{by (BF) applied to ``.''}\\
m/\langle \mathtt{m}, 1\rangle\langle ., 1\rangle &= m/\langle =_3, 2\rangle\langle ., 1\rangle && \text{by (BV) applied to } \mathtt{A}\\
m/\langle \mathtt{m}, 1\rangle\langle ., 2\rangle &= m/\langle \mathtt{m}, 1\rangle && \text{by (BV) applied to } \mathtt{X}\\
m/\langle \mathtt{m}, 2\rangle &= m/\langle \mathtt{m}, 2\rangle && \text{by (BV) applied to } \mathtt{Y}\\
m/\langle \mathtt{m}, 3\rangle &= m/\langle =_3, 1\rangle && \text{by (BV) applied to } \mathtt{Z0}\\
m/\langle =_3, 2\rangle\langle ., 2\rangle &= \overline{m/\langle \mathtt{m}, 3\rangle} && \text{by (BV) applied to } \mathtt{Z}
\end{aligned}
$$

From the entire definition, we obtain 24 constraints. Elimination of the constraints on $=_k$, however, leaves only four constraints:

$$
\begin{aligned}
m(\langle \mathtt{m}, 1\rangle) &= in, & m/\langle \mathtt{m}, 1\rangle\langle ., 2\rangle &= m/\langle \mathtt{m}, 1\rangle,\\
m/\langle \mathtt{m}, 2\rangle &= m/\langle \mathtt{m}, 1\rangle, & m/\langle \mathtt{m}, 3\rangle &= \overline{m/\langle \mathtt{m}, 1\rangle}.
\end{aligned}
$$

We could regard the above set of constraints itself as representing the principal mode of the program, but the principal mode can be represented more explicitly in terms of a mode graph (Figure 3.2). Mode graphs are a kind of features graphs (feature structures with cycles) [1] in which

1. paths represent paths in P_{Atom},
2. the node corresponding to the path p represents the value $m(p)$,
3. arcs are labelled with the pair $\langle symbol, arg\rangle$ of predicate/function symbols and argument positions, and may have "negative signs" (denoted "•" in Figure 3.2) that invert the interpretation of the mode values of the paths beyond those arcs, and
4. binary constraints of the forms $m/p_1 = m/p_2$ and $m/p_1 = \overline{m/p_2}$ are represented by the sharing of nodes.

Mode analysis proceeds by merging many simple mode graphs representing individual mode constraints. Thus its decidability is guaranteed by the decidability of the

unification algorithm for feature graphs. The principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary (i.e., constraint on the mode value of, or the submode at, a particular path) or binary (i.e., constraint between the submodes at two particular paths).

Rule (GV) in Figure 2.1 contains a conditional mode constraint. However, we assume that guard goals are calls to built-in predicates whose mode graphs have been obtained beforehand. Thus the constraints actually imposed by Rule (GV) will be of the unary form $m(p) = in$ or $m/p = IN$. The rule will become simpler by allowing polymorphic guard goals, as will be discussed in Section 3.6.

Rule (BV) may impose constraints between three or more constraints, which cannot be represented as mode graphs by themselves. However, by *delaying* them, most of them can be reduced to unary/binary ones by other constraints [37]. Theoretically, some non-binary constraints may remain unreduced, for which it is most practical to let programmers declare the submodes of relevant paths.

The cost of mode analysis is almost proportional to the size n of the program and to the size d of the subgraph of the entire mode graph rooted at each predicate argument [46]. The size d reflects the complexity of communication protocols used in the program. To be precise, the time complexity is $O(nd \cdot \alpha(n))$, where α is the inverse of the Ackermann function.

We have analyzed various GHC/KL1 programs including the mode analyzer itself [37]. We have observed that, although larger programs have larger mode graphs because they use more predicate symbols, the value of d does not become so large (say several tens of nodes) even for programs using quite complicated communication protocols. Thus we expect that the mode graphs of very large programs are, in general, wide and shallow, which is to say most nodes can be reachable within several steps from the root.

3.3 Non-Well-Moded Programs

Programs that do not have well-modings are called non-well-moded. Since Moded Flat GHC programs must observe the principle of cooperative communication, non-well-modedness indicates that the communication protocols specified by the program are faulty. For instance, the following quicksort program is non-well-moded:

```
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
qsort([],     Ys0,Ys ) :- true | Ys =1 Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
   part(X,Xs,S,L), qsort(S,Ys0,Ys1), Ys2 =2 [X|Ys1], qsort(L,Ys2,Ys3).
   (the unification goal should have been Ys1 =2 [X|Ys2])
```

The first clause imposes the constraint (among others)

(1) $m(\langle \texttt{qsort}, 3\rangle) = in$ by (BF) applied to "[]",

while the second clause imposes

(2) $m/\langle =_1, 1\rangle = m/\langle \texttt{qsort}, 3\rangle$ by (BV) applied to Ys,

(3) $m/\langle =_1, 2\rangle = \overline{m/\langle =_1, 1\rangle}$ by (BU) applied to $=_1$,

(4) $m/\langle \texttt{qsort}, 2\rangle = m/\langle =_1, 2\rangle$ by (BV) applied to Ys0.

The third clause is first normalized to

```
qsort([X|Xs],Ys0,Ys3):- true |
  part(X,Xs,S,L),qsort(S,Ys0,Ys1),qsort(L,[X|Ys1],Ys3).
```

and imposes

(5) $m/\langle \texttt{qsort}, 2\rangle = in$ by (BF) applied to"|".

The set of those five constraints are inconsistent. Constraints (1)–(4) together entail $m(\langle \texttt{qsort}, 2\rangle) = out$, which is clearly inconsistent with Constraint (5).

We consider explaining the reasons of mode errors in terms of minimal inconsistent subsets of the set of mode constraints, because such subsets will be useful for locating errors. If we find multiple inconsistent subsets that are disjoint, they are considered as indicating different bugs in the program. Thus the technique can be used for locating multiple bugs at once.

Let us get back to the quicksort example. Observe that all proper subsets of Constraints (1)–(5) are consistent. Hence the five constraints form a minimal inconsistent subset. Since the quicksort program imposes 53 constraints in total, including constraints from the predicate part, we have succeeded in finding an adequately small subset.

Once a minimal inconsistent subset is found, how can one pinpoint a bug? It is reasonable in a moded framework to assume that programmers have *intended modes* of their programs (though not declared explicitly). In the above example, the programmer should be able to find that Constraint (5) is wrong because the second argument of qsort is intended to return the result of sorting. The analyzer tells what symbol occurrence in what clause imposes that constraint, which is the exact location of the bug.

A programmer may not always find it easy to tell whether each single constraint in a minimal subset conforms to the intended mode. However, the analyzer can present various consequences of a minimal subset of constraints as follows: Suppose the minimal subset S contains constraints from n (> 1) clauses. Then, it can be divided into n disjoint subsets $S_1, \ldots, S_n$ based on what clauses imposed what constraints. Because S is minimal, each of $S_1, \ldots, S_n$ is consistent. So we can form a mode graph for each S_i to make the consequences entailed by S_i explicit and find what clause contains a bug.

The mode constraints imposed by a program usually have redundancy. That is, a single bug could be explained by many possible minimal inconsistent subsets. However, a subset corresponding to a local portion of a program is likely to be a better explanation than a subset corresponding to a larger or scattered portion, because it reflects the program structure better and facilitates debugging. So, after describing basic algorithms in Section 3.4, we consider in Section 3.5 how to divide programs into layers based on the structure of process definitions and search minimal subsets locally within each process definition.

3.4 Finding Minimal Subsets

We consider various algorithms for finding a minimal inconsistent subset of a inconsistent set of mode constraints $C = \{c_1, \ldots, c_n\}$. This section presents simple algorithms for finding a single inconsistent subset and extend them to find multiple disjoint subsets.

3.4.1 The Basic Algorithms

Let $C = \{c_1, \ldots, c_n\}$ be an inconsistent set of constraints. Algorithm 1 below finds a single minimal inconsistent subset from C. In the algorithm, the merging of constraint sets and the checking of consistency are realized as the unification of mode graphs and the checking of its success/failure. Although the algorithm is quite general, its efficiency hinges upon the fact that there is a pair of efficient algorithms for computing the union of constraint sets and checking its consistency.

- **Algorithm 1**

  $S \leftarrow \{\}$;
  while S is consistent **do**
    $D \leftarrow S$; $i \leftarrow 0$;
    while D is consistent **do**
      $i \leftarrow i + 1$; $D \leftarrow D \cup \{c_i\}$
    end;
    $S \leftarrow S \cup \{c_i\}$
  end

The set S thus obtained is a minimal inconsistent subset. To see why, let $S_j/D_j/i_j$ be the values of $S/D/i$ at the end of the jth iteration of the outer loop, and k be the size of S. It suffices to show that the set $S_k \setminus \{c_{i_j}\} = \{c_{i_1}, \ldots, c_{i_{j-1}}, c_{i_{j+1}}, \ldots, c_{i_k}\}$ is consistent for any i_j, but it is easy to see that

- $D_j \setminus \{c_{i_j}\}$ is consistent (because the inner loop was not exited when $D = D_j \setminus \{c_{i_j}\}$), and

- $S_k \backslash \{c_{i_j}\} = \{c_{i_1}, \ldots, c_{i_{j-1}}, c_{i_{j+1}}, \ldots, c_{i_k}\} \subseteq D_j \backslash \{c_{i_j}\}$, because
  - $D_j \backslash \{c_{i_j}\} = S_{j-1} \cup \{c_1, c_2, \ldots, c_{i_j - 1}\}$,
  - $S_{j-1} = \{c_{i_1}, c_{i_2}, \ldots, c_{i_{j-1}}\}$, and
  - $\{c_{i_{j+1}}, \ldots, c_{i_k}\} \subseteq \{c_1, c_2, \ldots, c_{i_j - 1}\}$ (because $i_1 > i_2 > \ldots > i_k$).

Hence all proper subsets of S are consistent.

Now we consider the complexity of the above algorithm. As explained in Section 3.2, it takes $O(nd \cdot \alpha(n))$ time to merge n mode constraints. The time complexity of finding a minimal subset with k elements out of n mode constraints is $O(nkd \cdot \alpha(n))$, because

- in each iteration, we must merge at most n constraints until inconsistency arises, and
- it takes k iterations until a minimal subset with k elements is obtained.

Usually, k is a small value independent of the program size, as we will see in Section 3.7.

A variant of the above algorithm will compute a better minimal subset. Let $C = \{c_1, \ldots, c_n\}$ be such that $i < j$ implies that the symbol (occurrence) imposing c_i occurs textually before the symbol (occurrence) imposing c_j. Then it is likely that a minimal subset can be formed from a rather small range of the sequence $c_1, \ldots, c_n$, and such a local subset is considered a good explanation. If this is the case, scanning S in alternate directions will be more efficient and compute a better solution:

- **Algorithm 1′**

```
S ← {}; i ← 0; j ← 1;
while S is consistent do
   D ← S;
   while D is consistent do
      i ← i + j; D ← D ∪ {c_i}
   end;
   S ← S ∪ {c_i}; j ← −j
end
```

3.4.2 Finding Multiple Independent Minimal Subsets

Algorithms 1 and 1′ compute a single minimal inconsistent subset S of C. To compute multiple, independent minimal subsets, we can simply re-apply the algorithms after removing the elements of S from C. This enables the analyzer to detect as many independent bugs as possible at once. Note that Algorithm 2 below uses a self-contradictory constraint as a sentinel.

- **Algorithm 2**

```
c_0 ← false;
while true do
    let c_1, ..., c_m be the elements of C;
    i ← m + 1; j ← −1; S ← {};
    while S is consistent do
        D ← S;
        while D is consistent do
            i ← i + j; D ← D ∪ {c_i}
        end;
        S ← S ∪ {c_i}; j ← −j
    end;
    if i = 0 then exit
    else
        report(S); C ← C\S
    fi
end
```

3.5 Diagnosing Stratified Programs

Algorithms in Section 3.4 find minimal subsets from the entire set of constraints without reference to the logical structure of the programs to be analyzed. However, the set of constraints can be very large. If we divide the set of constraints taking the problem domain (= program analysis) into account and analyze the obtained subsets separately, we may be able to reduce the amount of computation and obtain more useful information for debugging.

3.5.1 Call Graphs and Process Graphs

When dividing Flat GHC programs according to their logical structures, clauses defining a concurrent process by means of self or mutual recursion can be considered to form a *process definition.*

A program defines a directed graph, called a *call graph*, that describes the caller-callee relationship between predicates. A call graph is a directed graph such that each node v corresponds to a Flat GHC predicate, and an arc e from a node v to a node v' means that the predicate v calls the predicate v' directly from a clause body.

The strongly connected components of a call graph exactly correspond to process definitions in the above sense. Processes defined by mutual recursion are naturally recognized in this way. A non-recursive predicate that spawns one or more subprocesses is regarded as a process by itself.

It is well-known that division into strongly connected components is uniquely determined and can be done in $O(n + a)$ time, where n is the number of nodes and a is the number of arcs in the graph.

Division into strongly connected components is regarded as the division of graph nodes into equivalence classes. The quotient graph obtained by contracting the arcs inside strongly connected components is called a *process graph*. A process graph is a dag representing the dependency relation between processes. Henceforth we confuse a node of a process graph with the process definition represented by the node.

3.5.2 Program Stratification

Since the process graph G of a program is acyclic, the partial order defined by G can be used for stratifying the program. We define the layer number $L(v)$ of the node v as:

$$L(v) = \max(\{L(v') \mid v' \in Adj^+(v)\} \cup \{0\}) + 1 \ ,$$

where $Adj^+(v)$ means the set of destination nodes of the arcs from v. Note that the above definition assigns 1 to nodes without outgoing arcs.

3.5.3 Finding Relative Minimal Subsets

Bugs of stratified programs can be classified into (1) those within each layer and (2) those across layers. Since bugs of the first kind can be found simply by checking each node of a process graph independently, we consider how to deal with bugs of the latter kind, assuming that each process definition is well-moded.

How to find a minimal subset from a stratified program depends on how we consider non-well-modedness across layers. We adopt bottom-up analysis, that is, we choose to check process definitions from lower layers (those with smaller numbers). Bottom-up analysis lends itself to the analysis of large programs that may use existing program libraries.

Suppose bottom-up analysis has found an inconsistency in an attempt to merge constraints from the kth layer and those from lower layers. Since each process definition in the kth layer is consistent by assumption and different process definitions in the same layer are independent, the kth layer itself is consistent. Hence the reason of inconsistency can be attributed to either (or both) of the following:

- The kth layer wrongly uses the lower layers.
- The lower layers, though well-moded, have an unintended principal mode.

It is rather difficult to tell which, but a reasonable solution is to ask the programmer to check the kth layer first before suspecting lower layers. This is reasonable because a concise explanation should be considered first. Locating bugs inside well-moded layers

is somewhat beyond the principal scope of mode analysis, though their mode graphs will provide useful information.

Bottom-up analysis considerably limits search space by finding from a process definition a minimal subset of constraints that are inconsistent with the set B of constraints from lower layers. Such a subset is called a *minimal subset relative to* B. In the following algorithm, $C(v)$ initially holds the set of constraints imposed by the node v of the process graph.

- **Algorithm 3**

  for $k \leftarrow 1$ **to** the highest layer number **do**
    for each v **in** $\{v \mid L(v) = k\}$ **do**
      $B \leftarrow \bigcup_{v' \in Adj^+(v)} C(v')$;
      apply Algorithm 2′ (shown below);
      $C(v) \leftarrow B \cup C(v)$
    end
  end

Algorithm 2′ reports and removes minimal inconsistent subsets of the set $C(v)$ of constraints relative to B:

- **Algorithm 2′**

  $c_0 \leftarrow$ false;
  while true **do**
    let $c_1, \ldots, c_m$ be the elements of $C(v)$;
    $i \leftarrow m+1$; $j \leftarrow -1$; $S \leftarrow B$;
    while S is consistent **do**
      $D \leftarrow S$;
      while D is consistent **do**
        $i \leftarrow i + j$; $D \leftarrow D \cup \{c_i\}$
      end;
      $S \leftarrow S \cup \{c_i\}$; $j \leftarrow -j$
    end;
    if $i = 0$ **then exit**
    else
      $S \leftarrow S \backslash B$;
      report(S); $C(v) \leftarrow C(v) \backslash S$
    fi
  end

In an actual implementation, S and D are represented during iterations as mode graphs which are destructively updated by the set union operations. The results to be reported should be represented again by the set of constraints, but this can be obtained efficiently by recording what c_i's have been added to S by the assignment $S \leftarrow S \cup \{c_i\}$.

3.6 Stratification and Mode Polymorphism

When two or more processes share a process definition in a lower layer, Algorithm 3 may cause a problem.

Consider a program that uses a "generic" (or polymorphic) predicate such as stream merging (Section 3.2) in various modes. When such a predicate is called from different places, the process graph will contain a shared node. Suppose a process definition at the node v and another definition at v' use a predicate p polymorphically. Then the analyses of the node v and of v' will succeed because they are independent, but the analysis of a higher-level node that uses both v and v' will detect the inconsistent use of p as an error.

However, we can regard predicates at lower layers as polymorphic when called from higher layers. To allow stratification-based polymorphism, we need to create a copy of the mode graph of a polymorphic predicate for each call to that predicate. This can be achieved by indexing each polymorphic call (as we have done for unification goals) and creating a copy of the mode graph for each polymorphic call, modifying their paths according to the indices.

So, we assume that

1. for each polymorphic predicate p, the preprocessing phase numbers all calls to p from higher layers from 1 upwards, and
2. the first element of each path in P_{Atom} is of the form $\langle p_s, i \rangle$, where s is a sequence of natural numbers and can be omitted if empty.

Also, let $C_k(v)$ be a modified copy of the mode constraints $C(v)$ such that the first elements of the paths are changed from the form $\langle p_s, l \rangle$ to $\langle p_{sk}, l \rangle$.

- **Algorithm 3′**

  The same as Algorithm 3, except that the assignment $B \leftarrow \bigcup_{v' \in Adj^+(v)} C(v')$ is replaced by:

  $B \leftarrow \{\}$;
  for each indexed (i.e., polymorphic) body goal g **in** v **do**
  let k be the index of g;
  let v' be the node defining the polymorphic predicate;
  $B \leftarrow B \cup C_k(v')$
  end

Although we have focused on the stratification of predicates called from clause bodies, the same idea could be applied to test predicates called from guards. This is useful for introducing polymorphism to test predicates, under which Rule (GV) in Figure 2.1 does not have to use implication any more to avoid the "back propagation" of mode constraints to generic guard predicates.

3.7 Experiments

We have made some initial experiments to see how the basic algorithms help bug location and how large minimal subsets can be.

First, we applied Algorithm 1′ to 20 erroneous programs, each containing a single near-miss bug such as

1. "tell" unification specified in a clause head (as in Prolog) rather than in a body,
2. misspelling of a variable name, and
3. wrong order of arguments.

All those bugs will impose constraints inconsistent with those from correct clauses.

Sixteen of the 20 programs we used were small and imposed less than 100 constraints each, while the remaining four programs imposed about 500 constraints each. The sizes of the minimal inconsistent subsets varied from 2 to 8, with the average being 3.75. The sizes of minimal subsets were independent of the total number of constraints. Thus we have ascertained that the parameter k in the complexity measure in Section 3.4.1 is a rather small constant.

We have also ascertained that multiple bugs can be detected at once if they are not too close to each other. However, since our algorithm removes some correct constraints together with incorrect constraints when finding the first bug, it is possible that the second bug does not cause inconsistency any more. Fortunately, this will not happen so often because the mode constraints imposed by a program usually contain redundancy and the number of removed correct constraints is usually small.

We have not yet analyzed very large programs, but thanks to the constraint-based approach, large programs can (and will) be analyzed in smaller pieces. Stratification will automatically divide a program into pieces, too. Thus we can expect that our positive results will apply to larger programs quite well.

It would be unrealistic to search for *all* minimal inconsistent subsets covering a single bug because it requires much more computation. However, it will be less unrealistic to compute *several* inconsistent subsets which share some constraint. If the program contains a single bug, a constraint shared by all minimal subsets is likely to indicate

the bug. For instance, suppose we wrote the stream merging program as:

```
m([],Y,Z) :- true | Z =1 Y.
m(X,[],Z) :- true | Z =2 X.
m([A|X],Y,Z0) :- true | Z0 =3 [A|Z], m(X,Y,Z0).
(the final goal should have been m(X,Y,Z))
m(X,[A|Y],Z0) :- true | Z0 =4 [A|Z], m(X,Y,Z).
```

The mode analyzer first normalized the program, converting the third clause to

```
m([A|X],Y,Z0) :- true | Z0 =3 [A|Z], m(X,Y,[A|Z]).
```

and then found that it had at least four minimal subsets (with 5, 5, 4, and 4 elements). The only constraint included in all of those subsets was $m(\langle \mathtt{m}, 3 \rangle) = in$, which was imposed by Rule (BF) applied to the list constructor occurring in the third argument of the recursive goal of the (normalized) third clause. Thus we succeeded in pinpointing the exact location of the bug in this case. It is a subject of future work how to compute a sufficient number of overlapping subsets efficiently to pinpoint a bug.

3.8 Related Work

As mentioned in Section 1, most previous work on the mode analysis of (concurrent) logic languages was based on abstract interpretation, and focused mainly on the reasoning of program properties assuming that the programs were correct. In contrast, constraint-based mode analysis can be used for diagnosis as well as optimization by assuming that correct programs are well-moded.

Concurrent logic languages Doc [14] and Janus [25] let programmers distinguish between input and output occurrences using annotations. These annotations can be regarded as mode declarations, the consistency of which needs to be checked statically or dynamically. So the technique proposed in this chapter applies also to those languages. The purpose of the mode system of PARLOG is quite different, as discussed in [46].

Somogyi [28] proposed another framework of strong moding independently and studied its implications in depth. His framework shares some features with ours, such as the principle of cooperative communication and the capability of dealing with bidirectional communication. An advantage of our constraint-based framework is that, besides being simple, it provides a unified framework for mode declaration, mode checking and mode inference. This makes it realistic to analyze existing programs and still enables programmers to declare intended modes that can be used as correct mode constraints in finding minimal subsets.

Mercury [29] is another recent strongly moded language. Being a purely declarative logic language, however, its mode system is very different from the mode system of

Moded Flat GHC; the former deals with the change of instantiatedness, which is a temporal property, while the latter deals with polarity, which is a non-temporal property [37].

Chen et al. [4] proposed an algorithm for finding maximal unifiable subsets and minimal non-unifiable subsets of a set of equations. They use hypergraph structures that record the reasons of (non-)unifiability during unification. Our algorithms use mode graphs that do not retain reason information and reconstruct them repeatedly to find minimal subsets. Although this simple approach turned out to work quite well, it is a subject of future work to compare our approach with the hypergraph approach.

Analysis of malfunctioning systems based on their intended logical specification has been studied in the field of artificial intelligence [23] and known as model-based diagnosis. Model-based diagnosis has similarities with our work in the ability of searching minimal explanations and multiple faults. However, the purpose of model-based diagnosis is to analyze the differences between intended and observed behaviors. Our mode system does *not* require that the intended behavior of a program be given as mode declarations, and still locates bugs quite well.

3.9 Conclusions

We have proposed algorithms for diagnosing non-well-moded concurrent logic programs based on the searching of minimal inconsistent subsets of mode constraints. Once minimal subsets are found, it is straightforward for the system to indicate suspected symbol(occurrence)s in the program and/or to show logical consequences a (consistent) subset of the minimal inconsistent subsets entails. We have also shown how we can obtain "good" (i.e., local) explanations of mode errors by dividing programs based on their logical structures. All these techniques are very systematic for static error analysis and are efficient as well.

It is not realistic or helpful to search all minimal inconsistent subsets, but it might be reasonable to find several of them because, if some mode constraint is shared by all the minimal subsets, it is likely to indicate the exact location of a bug. This means to take advantage of the redundancy of mode constraints to guess the exact location of a bug. By pinpointing erroneous constraints this way, the ability of detecting multiple bugs will be improved further.

第4章　モード体系を利用した誤り訂正

We study how constraint-based static analysis can be applied to the automated and systematic debugging of program errors.

Strongly moding and constraint-based mode analysis are turning to play fundamental roles in debugging concurrent logic/constraint programs as well as in establishing the consistency of communication protocols and in optimization. Mode analysis of Moded Flat GHC is a constraint satisfaction problem with many simple mode constraints, and can be solved efficiently by unification over feature graphs. We have proposed a simple and efficient technique which, given a non-well-moded program, diagnoses the "reasons" of inconsistency by finding minimal inconsistent subsets of mode constraints. Since each constraint keeps track of the symbol occurrence in the program that imposed the constraint, a minimal subset also tells possible sources of program errors. The technique is quite general and can be used with other constraint-based frameworks such as strong typing.

Based on the above idea, we study the possibility of *automated debugging in the absence of mode/type declarations*. The mode constraints are usually imposed redundantly, and the constraints that are considered correct can be used for correcting wrong symbol occurrences found by the diagnosis. As long as bugs are near-misses, the automated debugger can propose a rather small number of alternatives that include the intended program. Search space is kept small because constraints effectively prune many irrelevant alternatives. We demonstrate the technique by way of examples.

4.1 Introduction

This chapter proposes a framework of automated debugging of program errors under static, constraint-based systems for program analysis, and shows how and why program errors can be fixed in the absence of programmers' declarations. The language we are particularly interested in is Moded Flat GHC [44][46] proposed in 1990. Moded Flat GHC is a concurrent logic (and consequently, a concurrent constraint) language with a constraint-based mode system designed by one of the authors, where modes prescribe the information flow that may be caused by the execution of a program.

Languages equipped with strong typing or strong moding[1] enable the detection of a type/mode errors by checking or reconstructing types or modes. The best-known

[1]Modes can be thought of as "types in a broad sense," but in this chapter we reserve the term "types" to mean sets of possible values.

framework for type reconstruction is the Hindley-Milner type system [20], which allows us to solve a set of type constraints obtained from program text efficiently as a unification problem.

Similarly, the mode system of Moded Flat GHC allows us to solve a set of mode constraints obtained from program text as a constraint satisfaction problem. Without mode declarations or other kinds of program specification given by programmers, mode reconstruction statically determines the read/write capabilities of variable occurrences and establishes the consistency of communication protocols between concurrent processes [46]. The constraint satisfaction problem can be solved mostly (though not entirely) as a unification problem over feature graphs (feature structures with cycles) and can be solved in almost linear time with respect to the size of the program [1]. As we will see later, types also can be reconstructed using a similar (and simpler) technique.

Compared with abstract interpretation usually employed for the precise analysis of program properties, constraint-based formulation of the analysis of basic properties has a lot of advantages. Firstly, thanks to its incremental nature, it is naturally amenable to separate analysis of large programs. Secondly, it allows simple and general formulations of various interesting applications including error diagnosis.

When a concurrent logic program contains bugs, it is very likely that mode constraints obtained from the erroneous symbol occurrences are incompatible with the other constraints. We have proposed an efficient algorithm that finds a minimal inconsistent subset of mode constraints from an inconsistent (multi)set of constraints [7]. A minimal inconsistent subset can be thought of as a minimal "explanation" of the reason of inconsistency. Furthermore, since each constraint keeps track of the symbol occurrence(s) in the program that imposed the constraint, a minimal subset tells possible sources (i.e., symbol occurrences) of program errors. Our technique can locate multiple bugs at once. The technique is quite general and can be used with other constraint-based frameworks such as strong typing.

Since the conception of the above framework of program diagnosis and some experiments, we have found that the multiset of mode constraints imposed by a program usually has redundancy and it usually contains more than one minimal inconsistent subset when it is inconsistent as a whole. Redundancy comes from two reasons:

1. A non-trivial program contains conditional branches or nondeterministic choices. In (concurrent) logic languages, they are expressed as a set of rewrite rules (i.e., program clauses) that may impose the same mode constraints on the same predicate.
2. A non-trivial program contains predicates that are called from more than one place, some of which may be recursive calls. The same mode constraint may be imposed by different calls.

We can often take advantage of the redundancies and pinpoint a bug (Sect. 4.3) by

assuming that redundant modes are correct. The next step worth trying is *automated error correction.* We can estimate the intended mode of a program from the parts of the program that are considered correct, and use it to fix small bugs, which is the main focus of this work.

Bugs that can be dealt with by automated correction are necessarily limited to near-misses, but still, automated correction is worth studying because:

- serious algorithm errors cannot be mechanically corrected anyway,
- if the algorithm for a program has been correctly designed, the program is usually "mostly correct" even if it doesn't run at all, and
- real-life programs are subject to a number of revisions, upon which small errors are likely to be inserted.

Our idea of error correction can be compared with error-correcting codes in coding theory. Both attempt to correct minor errors using redundant information. Unlike error-correcting codes that contain explicit redundancies, programs are usually not written in a redundant manner. However, programs interpreted in an abstract domain may well have *implicit* redundancies. For instance, the `then` part and the `else` part of a branch will usually compute a value of the same type, which should also be the same as the type expected by the reader of the value. This is exactly why the multiset of type or mode constraints usually has redundancies.

It is not obvious whether such redundancies can be used for automated error correction, because even if we correctly estimate the type/mode of a program, there may be many possible ways of error correction that are compatible with the estimated type/mode. The usefulness of the technique seems to depend heavily on the choice of a programming language and the power of the constraint-based static analysis. We have obtained promising results using Moded Flat GHC and its mode system, with the assistance of type analysis and other constraints.

The other concern in automated debugging is search space. Generate-and-test search, namely the generation of a possible correction and the computation of its principal mode (and type), can involve a lot of computation, but we can prune much of the search space by using 'quick-check' mode information to detect non-well-modedness. Types are concerned with aspects of program properties that are different from modes, and can be used together with modes to improve the quality of error correction.

4.2 Strong Moding and Typing in Concurrent Logic Programming

We first outline the mode system of Moded Flat GHC. The readers are referred to [46] and [37] for details.

In concurrent logic programming, modes play a fundamental role in establishing the safety of a program in terms of the consistency of communication protocols. The mode system of Moded Flat GHC gives a polarity structure (that determines the information flow of each part of data structures created during execution) to the arguments of predicates that determine the behavior of goals. A mode expresses this polarity structure, which is represented as a mapping from the set of *paths* to the two-valued codomain $\{in, out\}$. Paths here are strings of pairs, of the form $\langle symbol, arg \rangle$, of predicate/function symbols and argument positions, and are used to specify possible positions in data structures. Formally, the set P_{Term} of paths for terms and the set P_{Atom} of paths for atomic formulae are defined using disjoint union as:

$$P_{Term} = (\sum_{f \in Fun} N_f)^*, \; P_{Atom} = (\sum_{p \in Pred} N_p) \times P_{Term} \; ,$$

where $Fun/Pred$ are the sets of function/predicate symbols, and N_f/N_p are the sets of possible argument positions (numbered from 1) for the symbols f/p. The purpose of mode analysis is to find the set of all modes (each of type $P_{Atom} \to \{in, out\}$) under which every piece of communication is cooperative. Such a mode is called a *well-moding*. Intuitively, *in* means the inlet of information and *out* means the outlet of information. A program does not usually define a unique well-moding but has many of them. So the purpose of mode analysis is to compute the set of all well-modings in the form of a *principal* (i.e., most general) mode. Principal modes can be expressed naturally by mode graphs, as described later in this section.

Given a mode m, we define a *submode* m/p, namely m viewed at the path p, as a function satisfying $(m/p)(q) = m(pq)$. We also define IN and OUT as submodes returning *in* and *out*, respectively, for any path. An overline '$-$' inverts the polarity of a mode, a submode, or a mode value.

A Flat GHC program is a set of clauses of the form h :- G | B, where h is an atomic formula and G and B are multisets of atomic formulae. Constraints imposed by a clause h :- G | B are summarized in Fig. 4.1. Rule (BU) numbers unification body goals because the mode system allows different body unification goals to have different modes. This is a special case of mode polymorphism that can be introduced into other predicates as well [7], but in this chapter we will not consider general mode polymorphism because whether to have polymorphism is independent of the essence of this work.

For example, consider a quicksort program defined as in Figure 4.2. From the entire definition, we obtain 53 constraints which are consistent. We could regard these constraints themselves as representing the principal mode of the program, but the principal mode can be represented more explicitly in terms of a mode graph (Fig. 4.3). Mode graphs are a kind of feature graphs [1] in which

1. a path (in the graph-theoretic sense) represents a member of P_{Atom},
2. the node corresponding to a path p represents the value $m(p)$ ($\downarrow = in$, $\uparrow = out$),

(HF) $m(p) = in$, for a function symbol occurring in h at p.

(HV) $m/p = IN$, for a variable symbol occurring more than once in h at p and somewhere else.

(GV) If some variable occurs both in h at p and in G at p',
$\forall q \in P_{Term}(m(p'q) = in \Rightarrow m(pq) = in)$.

(BU) $m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}$, for a unification body goal $=_k$.

(BF) $m(p) = in$, for a function symbol occurring in B at p.

(BV) Let v be a variable occurring exactly $n\,(\geq 1)$ times in h and B at $p_1, \ldots, p_n$, of which the occurrences in h are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & \text{if } k = 0; \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & \text{if } k > 0; \end{cases}$$

where the unary predicate $\mathcal{R}$ over finite *multisets* of submodes represents "cooperative communication" between paths and is defined as

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S(s(q) = out \wedge \forall s' \in S \backslash \{s\}\, (s'(q) = in)).$$

Fig. 4.1: Mode constraints imposed by a program clause h :- G | B or a goal clause :- B.

```
quicksort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
qsort([],     Ys0,Ys ) :- true | Ys =1 Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
   part(X,Xs,S,L), qsort(S,Ys0,Ys1), Ys1 =2 [X|Ys2], qsort(L,Ys2,Ys3).
part(_,[],     S, L ) :- true | S =3 [], L =4 [].
part(A,[X|Xs],S0,L ) :- A>=X | S0 =5 [X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0 =6 [X|L], part(A,Xs,S,L).
```

Fig. 4.2: A quicksort program.

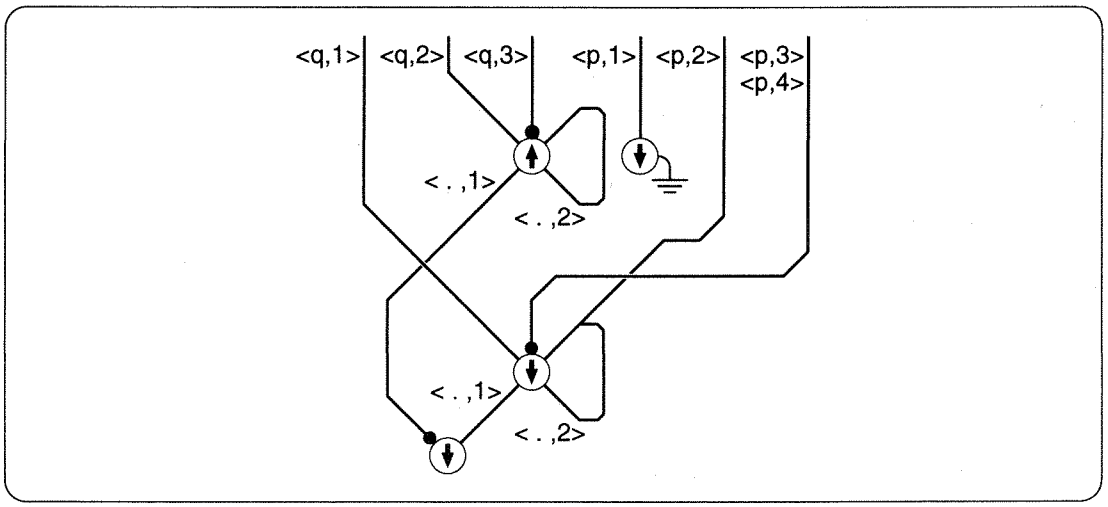

Fig. 4.3: The mode graph of a quicksort program. `q` stands for `qsort` and `p` stands for `part`. The mode information of the toplevel predicate and unification goals is omitted.

3. each arc is labeled with the pair $\langle symbol, arg \rangle$ of a predicate/function symbol and an argument position, and may have a "negative sign" (denoted "•" in Fig. 4.3) that inverts the interpretation of the mode values of the paths beyond that arc, and
4. a binary constraint of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$ is represented by letting p_1 and p_2 lead to the same node.

Mode analysis proceeds by merging many simple mode graphs representing individual mode constraints. Thus its decidability is guaranteed by the decidability of the unification algorithm for feature graphs. The principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary (i.e., constraint on the mode value of, or the submode at, a particular path) or binary (i.e., constraint between the submodes at two particular paths). Space limitations do not allow us to explain further details, which can be found in [37].

A type system for concurrent logic programming can be introduced by classifying a set *Fun* of function symbols into mutually disjoint sets $F_1, \ldots, F_n$. A type here is a function from P_{Atom} to the set $\{F_1, \ldots, F_n\}$. Like principal modes, principal types can be computed by unification over feature graphs. Constraints on a well-typing τ are summarized in Fig. 4.4. The choice of a family of sets $F_1, \ldots, F_n$ is somewhat arbitrary. This is why moding is more fundamental than typing in concurrent logic programming.

Mode and type analyses have been implemented as part of *klint*, a static analyzer for KL1 programs [39].

(HBF_τ) $\tau(p) = F_i$, for a function symbol occurring at p in h or B.

(HBV_τ) $\tau/p = \tau/p'$, for a variable occurring both at p and p' in h or B.

(GV_τ) $\forall q \in P_{Term}(m(p'q) = in \Rightarrow \tau(pq) = \tau(p'q))$, for a variable occurring both at p in h and at p' in G.

(BU_τ) $\tau/\langle =_k, 1\rangle = \tau/\langle =_k, 2\rangle$, for a unification body goal $=_k$.

Fig. 4.4: Type constraints imposed by a program clause h :- G | B or a goal clause :- B.

4.3 Identifying Program Errors

When a concurrent logic program contains an error, it is very likely (though not always the case) that its communication protocols become inconsistent and the set of its mode constraints becomes unsatisfiable. A wrong symbol occurring at some path is likely to impose a mode constraint inconsistent with correct constraints representing the intended specification.

A minimal inconsistent subset of mode constraints can be computed efficiently using a simple algorithm[2]. Let $C = \{c_1, \ldots, c_n\}$ be a multiset of constraints. Algorithm 1 below finds a single minimal inconsistent subset S from C when C is inconsistent. When C is consistent, the algorithm terminates with $S = \{\}$. $false$ is a self-inconsistent constraint used as a sentinel.

Algorithm 1:

$c_{n+1} \leftarrow false$;
$S \leftarrow \{\}$;
while S is consistent **do**
  $D \leftarrow S$; $i \leftarrow 0$;
  while D is consistent **do**
    $i \leftarrow i + 1$; $D \leftarrow D \cup \{c_i\}$
  end while;
  $S \leftarrow S \cup \{c_i\}$
end while;
if $i = n + 1$ **then** $S \leftarrow \{\}$

The readers are referred to [7] for a proof of the minimality of S, as well as various extensions of the algorithm. Note that the algorithm can be readily extended to finding multiple bugs at once. That is, once we have found a minimal subset covering a bug,

[2]The algorithm described here is a revised version of the one proposed in [7] and takes into account the case when C is consistent.

we can reapply the algorithm to the rest of the constraints.

In the algorithm, the merging of constraint sets and the checking of their consistency are realized mostly as the unification of mode graphs and the checking of its success/failure. Although the algorithm is quite general, its efficiency hinges upon the fact that there is a pair of efficient algorithms for computing the union of constraint sets and checking its consistency.

Our experiment shows that the average size of minimal inconsistent subsets is less than 4, and we have not yet found a minimal inconsistent subset with more than 11 elements. The size of minimal subsets turns out to be independent of the total number of constraints, and most inconsistencies can be explained by constraints imposed by a small range of program text.

Because we are dealing with near-misses, we can assume that most of the mode constraints obtained from a program represent an intended specification and that they have redundancies in most cases. In this case, one can often pinpoint a bug either

1. by computing a maximal consistent subset of size $n-1$ and taking its complement, or
2. by computing several overlapping minimal inconsistent subsets and taking their intersection.

Algorithm 2 described below combines these two alternative policies of pinpointing. To reduce the amount of computation, we do not compute all minimal subsets; instead, for each element (say s_i) of the initial inconsistent subset S, we execute Algorithm 1 after removing s_i from C, which will lead to another minimal subset if it exists. Thus Algorithm 2 simultaneously computes constraints suspected by the two policies.

Let $S = \{s_1, \ldots, s_m\}$ be a minimal subset obtained by Algorithm 1, and `getminimal`(C) be a function which computes a minimal inconsistent subset from a multiset C of constraints using Algorithm 1 above:

Algorithm 2:

$T \leftarrow S$;
for $j \leftarrow 1$ **to** m **do**
  $S' \leftarrow$ `getminimal`$(C \backslash \{s_j\})$;
  if $S' = \{\}$ **then**
    output $\{s_j\}$ as a solution of Policy 1
  else $T \leftarrow T \dot{\cup} S'$;
end for

Here, T is a *multiset* of constraints what serves as counters of the numbers of constraints occurring in S and (various versions of) S', and $\dot{\cup}$ is a multiset union operator. T records how many times each constraint occurred in different minimal subsets. Under Policy 2, constraints with more occurrences in T are more likely to be related to the source of the error.

Algorithm 2 is useful in locating multiple bugs at once. That is, once we have obtained a minimal inconsistent subset S, we can apply Algorithm 2 to refine the subset and remove only those constraints in the refined subset from C.

When Policy 1 outputs a single constraint imposed by an erroneous symbol occurrence, we need not consider Policy 2. However, there are cases where Policy 1 outputs no constraints or more than one constraint, in which case Policy 2 may better tell which constraints to suspect first.

Algorithm 2 is not always able to refine the initial set S, however. For instance, when S is the only minimal inconsistent subset, the algorithm will output all the elements of S by Policy 1 and will find no alternative subset by Policy 2. Fortunately, this is not a serious problem because S is usually quite small.

4.4 Automated Debugging Using Mode Constraints

Constraints that are considered wrong can be corrected by

- replacing the symbol occurrences that imposed those constraints by other symbols, or
- when the suspected symbols are variables, by making them have more occurrences elsewhere (cf. Rule (BV) of Fig. 2.1).

In this work, we focus on programs with a small number of errors in variables and constants; that is, we focus on errors in terminal symbols in abstract syntax trees. This may seem restrictive, but concurrent logic programs have quite flat syntactic structures (compared with other languages) and instead make heavy use of variables. Our experience tells that a majority of simple program errors arise from the erroneous use of variables, for which the support of a static mode system and debugging tools are invaluable.

An algorithm for automated correction is basically a search procedure whose initial state is the erroneous program, whose operations are the rewriting of the occurrences of variables or constants, and whose final states are well-moded programs[3]. This can be regarded also as a form of *abductive reasoning* which, from a presumably correct mode constraint B and the moding rules of the form "if A then B" (or "B for A") as shown in Fig. 2.1, infers a syntactic constraint A that is considered correct.

The symbols to be substituted in the correction are chosen from the constants or other variables occurring in the same clause. When the symbol to be rewritten occurs in the head, we should also consider replacement by a fresh variable. We don't have to try to form the mode graphs of all the alternative programs; from the set $C \setminus S$, we can derive a *replacement guideline*, namely simple constraints to be satisfied by the

[3]Here, we assume that errors can be corrected without changing the shape of the abstract syntax tree, though we could extend our technique and allow occurrences of terminal symbols to be simply added or deleted.

substituted symbol. Any replacement that violates the guideline will not lead to a well-moded program and can be rejected immediately.

Error correction may require the rewriting of more than one symbol occurrence. We perform iterative-deepening search with respect to the number of rewritings, because the assumption of near-misses implies that a simpler correction is more likely to be the intended one. These ideas have been partially implemented in the *kima* analyzer for KL1 programs [38].

4.5 Using Constraints Other Than Modes

When error correction requires the rewriting of more than one symbol occurrence, the iterative-deepening search may report a large number of alternative solutions, though they always include an intended one.

Using both the mode system and the type system reduces the number of alternatives greatly. Modes and types capture different aspects of a program, and rather few of well-formed programs are both well-moded and well-typed. We can expect that there are only a small number of well-moded and well-typed program syntactically in the 'neighborhood' of the given near-miss program.

The reason why a type system alone is insufficient should become clear by considering programs that are simple in terms of types such as numerical programs. The mode system is sensitive to the number of occurrences of variables (rule (BV) in Fig. 2.1) and can detect many errors that cannot be found by type analysis. However, even when the programs are simple in terms of types, types can be useful for inferring what constant should replace the wrong symbol.

Other heuristics from our programming experiences can reinforce the framework as well:

1. A singleton variable occurring in a clause body is highly likely to be an error.
2. A solution containing a variable occurring more than once in a clause head is less likely to be an intended one.

These heuristics are not as *ad hoc* as it might look; indeed they can be replaced by a unified rule on *constraint strength*:

- A well-moded solution with weaker mode constraints is more likely to be an intended one.

A singleton variable occurring at p in a clause body imposes a constraint $m/p = OUT$, which is much stronger than $m(p) = out$. Similarly, a variable occurring more than once at p_1, p_2, ... in a clause head imposes a constraint $m/p_i = IN$.

We could use more surface-level heuristics such as the similarity of variable names, but this is outside the scope of this work.

4.6 Experiments and Examples

We show some experimental results and discuss two examples of automated debugging. The examples we use are admittedly simple but that can be justified. First, we must anyway start with simple examples. Second, we have found that most inconsistencies can be explained by constraints imposed by a small range of program text, as we pointed out in Sect. 4.3. So we strongly expect that the total program size does not make much difference in the performance or the quality of automated debugging.

4.6.1 Experiments

We applied the proposed technique to programs with one mutation in variable occurrences. We systematically generated near-misses (each with one wrong occurrence of a variable) of three programs (there are many ways of inserting a bug) and examined how many of them became non-well-moded, whether automated correction reported an intended program, and how many alternatives were reported. Table 4.1 shows the results. In the table, the column "total cases" shows the numbers of cases examined, and the column "detected cases" shows how many cases lead to non-well-moded programs. For non-well-moded programs, we examined how many well-moded alternatives were proposed by the automated debugger by depth-1 search. In this experiment, we did not apply Algorithm 2 to refine a minimal inconsistent subset.

The programs we used are list concatenation (append), the generator of a Fibonacci sequence, and quicksort. We used the definitions of predicates only, that is, we did not use the constraints that might be imposed by the caller of these programs.

The row "mode only" indicates the results using mode constraints only, except that when correcting errors we regarded singleton variables in clause bodies as erroneous. In this experiment, minimal inconsistent subsets, when found, always included constraints imposed by the wrong symbol occurrence, and the original, intended programs were always included in the sets of the alternatives proposed by the algorithm.

A bug due to a wrong variable occurrence often results from misspelling (say the confusion of `YS` and `Ys`), in which case the original variable is likely to be replaced by a variable not occurring elsewhere in the clause. The row "new variable" shows the statistics of this case, which tells most errors were detected by mode analysis.

The row "mode & type" shows the improvement obtained by using types as well. The column "detected cases" shows that some of the well-moded erroneous programs were newly detected as non-well-typed. Note that the experiments did not consider the automated correction of well-moded but non-well-typed programs. For `fibonacci` and `quicksort`, we assumed that integers and list constructors belonged to different types. For `append`, we employed a stronger notion of types and assumed that the type of the elements of a list could not be identical to the type of the list itself.

The results show that the use of types was effective in reducing the number of alternatives. More than half of non-well-moded near-misses were uniquely restored to the

Table 4.1: Single-error detection and correction.

Program	Analysis	Total cases	Detected cases	Proposed alternatives 1	2	3	4	5	6	7	≥8
append	mode only	57	33	16	4	1	0	5	4	2	1
	new variable	13	11	7	0	0	1	1	2	0	0
	mode & type	57	44	19	3	2	5	1	3	0	0
fibonacci	mode only	84	43	28	7	0	0	0	2	3	3
	new variable	15	14	6	3	0	0	0	2	2	1
	mode & type	84	57	34	2	0	2	2	3	0	0
quicksort	mode only	245	148	84	33	2	3	1	8	7	10
	new variable	45	43	24	2	0	3	2	4	3	5
	mode & type	245	189	93	33	5	9	0	5	2	1

original program. Thus, programmers can benefit much from the support of constraint-based static analysis by writing programs in a well-moded and well-typed manner.

4.6.2 Example 1 — Append

As an example included in the above experiment, we discuss an append program with a single error. This example is simple and yet instructive.

R_1 : `append([], Y,Z ) :- true | Y =`$_1$ `Z.`
R_2 : `append([A|Y],Y,Z0) :- true | Z0 =`$_2$ `[A|Z], append(X,Y,Z).`
(The head should have been `append([A|X],Y,Z0)`)

Algorithm 1 computes the following minimal inconsistent subset of mode constraints:

	Mode constraint	Rule	Source symbol
(a)	$m/\langle \mathsf{append}, 1\rangle\langle ., 2\rangle = IN$	(HV)	`Y` in R_2
(b)	$m/\langle \mathsf{append}, 1\rangle = OUT$	(BV)	`X` in R_2

This tells that we should suspect the variables `X` and `Y` in Clause R_2. The search first tries to rewrite one of the occurrences of these variables (iterative-deepening), and finds six well-moded alternatives:

(1) R_2 : `append([A|X],Y,Z0) :- true | Z0 =`$_2$ `[A|Z], append(X,Y,Z).`
(2) R_2 : `append([A|Y],X,Z0) :- true | Z0 =`$_2$ `[A|Z], append(X,Y,Z).`
(3) R_2 : `append([A|Y],Y,Z0) :- true | Z0 =`$_2$ `[A|Z], append(Y,Y,Z).`
(4) R_2 : `append([A|Y],Y,Z0) :- true | Z0 =`$_2$ `[A|Z], append(Z0,Y,Z).`
(5) R_2 : `append([A|Y],Y,Z0) :- true | Z0 =`$_2$ `[A|Z], append(A,Y,Z).`
(6) R_2 : `append([A|Y],Y,Z0) :- true | Z0 =`$_2$ `[A|Z], append(Z,Y,Z).`

Types do not help much in this example, though Alternative (5) can be eliminated by an implicit type assumption described in Sect. 6.1 that list constructors and the elements of the list cannot occupy the same path. Alternatives (3), (4), (5) and (6) are programs that cause reduction failure for most input data, and can be regarded as less plausible solutions because of the two occurrences of `Y` in the clause heads that impose stronger constraints than intended.

What are Alternatives (1) and (2)? Alternative (1) is the intended program, and Alternative (2) is a program that merges two input lists by taking their elements alternately. It's not 'append', but is a quite meaningful program compared with the other alternatives!

In this example, Algorithm 2, if applied, will detect Constraint (b) as the unique result of Policy 1. This means that there must be some problems with the variable `X`, which in turn means that `X` must either be removed or occur more than once. Search of well-moded programs finds the same number of alternatives, but the search space is reduced because we do not have to consider the rewriting between `Y` and variables other than `X`.

4.6.3 Example 2 — Quicksort

Next, we consider a quicksort program with two errors.

```
1: R1 : quicksort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
2: R2 : qsort([],       Ys0,Ys  ) :- true | Ys =1 Ys0.
3: R3 : qsort([X|Xs],Ys0,Ys3) :- true |
4:         part(X,Xs,S,L),qsort(S,Ys0,Ys1),
5:         Ys2 =2 [X|Ys1],qsort(L,Ys2,Ys3).
```

(the unification should have been `Ys1` $=_2$ `[X|Ys2]`)

Algorithm 1 returns the following minimal inconsistent subset:

	Mode constraint	Rule	Source symbol
(a)	$m(\langle \texttt{qsort}, 3\rangle) = in$	(BF)	"`[]`" in R_1
(b)	$m/\langle =_1, 1\rangle = m/\langle \texttt{qsort}, 3\rangle$	(BV)	`Ys` in R_2
(c)	$m/\langle =_1, 2\rangle = \overline{m/\langle =_1, 1\rangle}$	(BU)	$=_1$ in R_2
(d)	$m/\langle \texttt{qsort}, 2\rangle = m/\langle =_1, 2\rangle$	(BV)	`Ys0` in R_2
(e)	$m(\langle =_2, 2\rangle) = in$	(BF)	"`.`" in R_3
(f)	$m/\langle =_2, 2\rangle = \overline{m/\langle =_2, 1\rangle}$	(BU)	$=_2$ in R_3
(g)	$m/\langle =_2, 1\rangle = \overline{m/\langle \texttt{qsort}, 2\rangle}$	(BV)	`Ys2` in R_3

This subset is inconsistent because two inconsistent constraints can be derived from it:

$$m(\langle \texttt{qsort}, 2\rangle) = out, \quad \text{by (a), (b), (c) and (d),}$$
$$m(\langle \texttt{qsort}, 2\rangle) = in, \quad \text{by (e), (f) and (g).}$$

It is worth noting that this example is rather difficult—the minimal subset is rather large and Algorithm 2 does not find an alternative minimal subset. That is, there is no redundancy of mode constraints in the formation of the difference list representing the result.

Thus we cannot infer the correct mode of the path $\langle \texttt{qsort}, 2 \rangle$ and other paths, and automated debugging should consider both of the possibilities, $m(\langle \texttt{qsort}, 2 \rangle) = in$ and $m(\langle \texttt{qsort}, 2 \rangle) = out$.

We consider the correction of both constants and variables here. It turns out that all depth-1 corrections are non-well-moded. There are six depth-2 corrections that are well-moded:

(1) Line 1: `quicksort(Xs,Ys) :- true | qsort(Xs,Zs,Zs).`
(2) Line 1: `quicksort(Xs,Ys) :- true | qsort(Zs,Ys,Zs).`
(3) Line 1: `quicksort(Xs,Ys) :- true | qsort(Xs,`c`,Ys).`
(4) Line 1: `quicksort(Xs,Ys) :- true | qsort(`c`,Ys,Xs).`
(5) Line 5: `Ys2` $=_2$ `[X|Ys2], qsort(L,Ys1,Ys3).`
(6) Line 5: `Ys1` $=_2$ `[X|Ys2], qsort(L,Ys2,Ys3).`

Here, c is some constant.

Typing doesn't help much for this example. The assumption that integers and list constructors should not occupy the same path does not exclude any of the above alternatives.

However, usage information will help. Suppose we know that `quicksort` is used as $m(\langle \texttt{quicksort}, 1 \rangle) = in$ and $m(\langle \texttt{quicksort}, 2 \rangle) = out$. This excludes Alternatives (1), (2) and (4). We can also exclude Alternative (5) by static occur-check (`Ys2` occurs on both sides of unification).

Of the remaining, Alternative (6) is the intended program that sorts items in ascending order. It is interesting to see that Alternative (3) is a program for sorting items in *descending* order by choosing '`[]`', the simplest element of the list type, as the constant c. This is not an intended program, but is a reasonable and approximately correct alternative which should not be rejected in the absence of program specification.

4.7 Related Work

Most previous work on the mode analysis of (concurrent) logic languages was based on abstract interpretation, and focused mainly on the reasoning of program properties assuming that the programs were correct. In contrast, constraint-based mode analysis can be used for diagnosis as well as optimization by assuming that correct programs are well-moded.

Analysis of malfunctioning systems based on their intended logical specification has been studied in the field of artificial intelligence [23] and known as model-based diagnosis. Model-based diagnosis has similarities with our work in the ability of searching

minimal explanations and multiple faults. However, the purpose of model-based diagnosis is to analyze the differences between intended and observed behaviors. Our mode system does *not* require that the intended behavior of a program be given as mode declarations, and still locates bugs quite well.

Wand proposed an algorithm for diagnosing non-well-typed functional programs [47]. His approach was to extend the unification algorithm for type reconstruction to record which symbol occurrence imposed which constraint. In contrast, our framework is built outside any underlying framework of constraint solving. We need not modify the constraint-solving algorithm but just call it. Besides its generality, our approach has an advantage that static analysis does not incur any overhead for well-moded/typed programs. Furthermore, the diagnosis guarantees the minimality of the explanation and often refines it further.

Comparison between Moded Flat GHC and other concurrent logic/constraint languages with some notions of moding can be found in [7].

4.8 Conclusions and Future Work

We studied how constraint-based static analysis could be applied to the automated and systematic debugging of program errors in the absence of mode/type declarations. We showed that, given a near-miss Moded Flat GHC program, our technique could in many cases report a unique solution or a small number of reasonable solutions that included the intended program.

If a programmer declares the mode and/or type of a program, that information can be used as constraints that are considered correct. In general, such constraints are useful in obtaining smaller minimal inconsistent subsets. However, our observation is that constraints implicitly imposed by the assumption of well-modedness (and well-typedness) is strong enough for automatic debugging to be useful.

It is a subject of future work to extend our framework to the correction of non-terminal program symbols (i.e., function and predicate symbols), mainly in terms of search space. It is yet to see whether the proposed framework works well for other programming paradigms such as typed functional languages and procedural languages, but we would claim that the concurrent logic/constraint programming paradigm benefits enormously from static mode/type systems.

第5章　処理系最適化への応用

The KLIC system has achieved both high portability and extensibility by employing C as an intermediate language and featuring generic objects that allow users to define new classes of data. It is also efficient for an untyped and unmoded language with fine-grained concurrency, but its flexibility incurs runtime overhead that could be reduced by static analysis. This chapter studies how constraint-based static analysis and abstract interpretation can be used to reduce dynamic data checking and to optimize loops. We applied the proposed technique to the optimization of floating-point numbers and their arrays. The optimized KL1 programs turned out to be only 34%–70% slower than the comparable C programs.

5.1 Introduction

The KLIC system [5] compiles KL1 [42] programs into efficient C programs, leaving low-level details of optimization to the underlying C compilers. Thus KLIC achieves high portability, and at the same time it is quite efficient as a symbolic processing language with tagged data representation. However, it is our belief that *parallel symbolic processing is fully justified only with static program analysis*, because the (costly) effort of parallelization is easily cancelled out by insufficient effort to improve single-processor performance. Also, real-life parallel symbolic applications (such as machine learning) may involve a lot of numerical computation. We anticipate that future symbolic languages should provide certain support of high-performance computing.

We developed a constraint-based mode system for GHC programs [46] and adapted it to deal with the features of KL1. The implementation of our mode analyzer is called *klint* [39]. The mode system deals with the direction of information flow and checks if every piece of communication is cooperative. A well-moded program is *unification-safe*, that is, it will not cause the failure of unification body goals (except due to occur-check). In the terminology of Concurrent Constraint Programming [24], this means that a *tell* operation will not make the constraint store inconsistent.

Constraint-based analysis can be used also for type analysis and linearity analysis, both partly supported by the current *klint* system [39]. Our implementation technique builds upon these three analyses, plus abstract interpretation of the instantiation states of variables, as described in Section 5.4.

In this chapter, we study how static program analysis improves the performance of the current KLIC implementation. Both for pedagogical and practical reasons, we take

simple numerical computation as an example to demonstrate the technique.

5.2 Number Representation in KLIC

KLIC employs 32-bit representation for scalar data. To accommodate tag bits, each integer is represented using the most significant 28 bits of a word, accompanied by a tag "0010". Floating-point numbers (64-bit long in KLIC) are implemented using *data objects*, one of the three kinds of generic objects [5]. A data object is represented as a pointer to a record containing the value (or a reference to the value) and a pointer to the table of methods available to the data. Vectors (one-dimensional arrays) are represented as data objects as well.

Although data objects are a natural and flexible means of defining new classes of data, accessing and operating on data objects involve the checking of tag values and the dereferencing of a couple of pointers. Another problem is that, since data objects are basically pointers to records, floating-point numbers not accessed any more become garbage. Integers do not have the garbage problem, but still involve the manipulation of tags—checking, removing (for arithmetic operations), and/or attaching.

Arrays that allow constant-time access and update are essential in many efficient algorithms. KLIC provides one-dimensional arrays called *vectors*. Vectors in a single-assignment language such as KL1 are necessarily immutable at the language level, so KLIC implements them as *multi-version data structures* in which old element values are preserved in an association list while the latest values are stored in random-access arrays [12]. In many programs, however, old element values thus preserved are not accessed later, in which case the management of an association list turns out to be an overhead. Also, KL1 vectors can store any values including uninstantiated variables, which means that before operating on a vector element, one must check if it is already instantiated and has the intended data type.

It is highly desirable to be able to identify a class of programs which, with static analysis, can be executed without tag operations for the checking of types and the availability of data. In addition to the compile-time techniques, we have designed and implemented an array class for *instantiated* numbers, which is intended to be used in combination with static analysis.

5.3 Constraint-Based Static Analysis

One of the novelties of our optimization technique is that it is largely based on *constraint-based* static analysis, which has a lot of advantages:

Simplicity. The mode system has been designed so that it is accessible to programmers. In other words, its purpose is not only for compilers to analyze programs but also for programmers to understand their programs better.

Efficiency. Mode analysis is a constraint satisfaction problem with many simple mode constraints, and can be solved efficiently by unification over feature graphs [1].

Modularity. Thanks to its incremental nature, it is naturally amenable to separate analysis of large programs.

Generality. It allows simple and general formulations of various interesting applications including the diagnosis of non-well-moded programs [7] and automated debugging [2].

We have designed three constraint-based systems for static analysis: mode, type, and linearity systems. Of these, the mode system provides the most fundamental information in the sense that it is referred to by type and linearity systems. Because mode analysis has been described in detail in the literature [46, 36, 37], we outline our type analysis and linearity analysis below.

5.3.1 Type Analysis

There may be a number of ways to introduce a type system into concurrent logic programming, but we chose to have a type system very similar in structure to the mode system. That is, a type tells what function symbols (including constant symbols) can occur at what positions in data structures. To mention a "position", we define the notion of a path as a sequence of pairs, of the form $\langle symbol, arg \rangle$, of predicate/function symbols and argument positions. Let P_{Atom} be the set of paths for specifying positions in a goal, and P_{Term} the set of paths for specifying positions in a term. For example, in a goal `p(f(a,b),C)`, the symbol `b` occurs at $\langle \mathtt{p}, 1 \rangle \langle \mathtt{f}, 2 \rangle \in P_{Atom}$. The set *Fun* of function symbols are assumed to be appropriately classified into mutually disjoint subsets $F_1, \ldots, F_n$. For instance, all integers may form one subset while all floating-point numbers may form another subset.

Formally, a type is a function from P_{Atom} to the set $\{F_1, \ldots, F_n\}$. Like principal modes, principal types can be computed by unification over feature graphs. A program and/or a goal clause is said to have a type τ if it satisfies all the typing constraints summarized in Figure 5.1. The choice of a family of sets $F_1, \ldots, F_n$ is somewhat arbitrary. This is another reason why moding is more fundamental than typing in concurrent logic programming.

5.3.2 Linearity Analysis

The purpose of linearity analysis (Chapter 6 and [40]) is to distinguish between data structures possibly referenced by two ore more pointers and those referenced by only one pointer. We call the former *shared* data and the latter *nonshared* data. Nonshared data structures can be recycled as soon as they are read by the sole reader (compile-time garbage collection).

(HBF$_\tau$) $\tau(p) = F_i$, for a function symbol in F_i occurring at p in h or B.

(HBV$_\tau$) $\forall q \in P_{Term}(\tau(pq) = \tau(p'q))$, for a variable occurring both at p and p' in h or B.

(GV$_\tau$) $\forall q \in P_{Term}(m(p'q) = in \Rightarrow \tau(pq) = \tau(p'q))$, for a variable occurring both at p in h and at p' in G.

(BU$_\tau$) $\forall q \in P_{Term}(\tau(\langle =_k, 1\rangle q) = \tau(\langle =_k, 2\rangle q))$, for a unification body goal $=_k$.

Fig. 5.1: Type constraints imposed by a program clause h :- G | B or a goal clause :- B.

Sharing of a data structure is caused by *nonlinear* variables defined as follows: An occurrence of a GHC variable is called a *channel occurrence* unless it is the second or subsequent occurrences in a clause head or an occurrence in a guard. A variable in a program clause or a goal clause is called *linear* if it has two or less channel occurrences in the clause, and *nonlinear* if it may have three or more channel occurrences. Mode analysis guarantees that exactly one of the channel occurrences of a variable is the writer occurrence and all the others are reader occurrences, so communication with a linear variable is one-to-one (or one-to-zero), while communication with a nonlinear variable is one-to-many. Readers are encouraged to see that surprisingly many of the variables in existing concurrent logic programs are linear.

To distinguish between nonshared data and shared data in structural operational semantics, we extend the semantics and attach either of the annotations 1 or ω to every function symbol f occurring in the bodies of program clauses and goal clauses. Suppose a substitution operation $\{v \leftarrow t\}$ is implemented by pointer assignment. Then $f^\omega(\ldots)$ means that the structure $f(\ldots)$ is possibly shared and $f^1(\ldots)$ means that $f(\ldots)$ is never shared. The annotations are managed as follows:

1. Annotations in program clauses and initial goal clauses are given according to how the structures are implemented. For instance, suppose two processes `p` and `q` in a goal clause

   ```
   :- p([1,2,3],X), q([1,2,3],Y).
   ```

   share a single list `[1,2,3]` in an *actual* implementation. Then the eight function symbols in the *textual* representation of the goal clause must have the annotation ω. If two separate lists are created at runtime, the annotations can be either 1 or ω. All function symbols occurring in a term with a principal function symbol with ω must have the annotation ω (closure condition).
2. In the extended operational semantics, the annotations are handled as follows.

(BF_λ) $\lambda(p) = \mathit{shared}$, for a function symbol f^ω occurring at p in B,

(LV_λ) For a linear variable with two occurrences at p_1 and p_2,

$$\forall q \in P_{\mathit{Term}}(m(p_1 q) = \mathit{in} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$$
(when p_1 is a head path)

$$\forall q \in P_{\mathit{Term}}(m(p_1 q) = \mathit{out} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$$
(when p_1 is a body path),

(NV_λ) For a nonlinear variable occurring at p (and elsewhere),

$\forall q \in P_{\mathit{Term}}(m(pq) = \mathit{out} \Rightarrow \lambda(pq) = \mathit{shared})$ (when p is a head path)

$\forall q \in P_{\mathit{Term}}(m(pq) = \mathit{in} \Rightarrow \lambda(pq) = \mathit{shared})$ (when p is a body path)

(BU_λ) For a unification body goal $=_k$,

$$\forall q \in P_{\mathit{Term}}(\langle =_k, 1\rangle q = \mathit{shared} \Leftrightarrow \langle =_k, 2\rangle q = \mathit{shared})$$

Fig. 5.2: Linearity constraints imposed by a program clause h :- G | B or a goal clause :- B.

Suppose an assignment $\{v \leftarrow t\}$ takes place in a goal reduction. This happens either by the execution of a unification goal $v = t$ or by the reduction of a non-unification goal $p(\ldots t \ldots)$ using a clause of the form $p(\ldots v \ldots)$:- G | B (renamed using fresh variables).

- When v is nonlinear, all the annotations in t are first changed to ω in order to indicate that multiple readers have (direct or indirect) access to all the subterms in t. Then the other occurrences of v (if any) are replaced by (modified) t.
- When v is linear, the other occurrence of v (if any) is replaced by t without changes of annotations.

The annotation can be viewed as modeling a 1-bit reference counter [6], though it is to be compiled away.

Linearity of a well-moded GHC program can be characterized by a linearity function $\lambda : P_{\mathit{Atom}} \rightarrow \{\mathit{nonshared}, \mathit{shared}\}$. A program clause h :- G | B or a goal clause :- B is said to have a linearity λ if it satisfies all the linearity constraints shown in Figure 5.2. The linearity constraints can be trivially satisfied by a function that always returns *shared*, but such a linearity function provides no useful information. The purpose of linearity analysis is to find which paths can have the value *nonshared*.

The main result about linearity analysis is the *Subject Reduction Theorem*, which guarantees that when a program $\mathcal{P}$ and a goal clause G satisfies λ and G is reduced to G' using $\mathcal{P}$, then G' also satisfies λ. It follows from this theorem that all data structures occurring at nonshared paths have an annotation 1.

5.4 Abstract Interpretation

Constraint-based program analysis nicely captures implementation-independent properties of programs, and provides basic information for optimization. However, it cannot capture time-dependent or implementation-dependent properties. An example of such properties is the instantiation state of the arguments of goals. It depends on both when it is observed (possibilities include when it is created, when it starts execution, and when it is finished) and how the goals are scheduled.

The analysis of instantiation states enables a compiler to form a *thread*, namely a sequence of (fine-grain) goals that can be executed sequentially without suspension checking (except upon entry into the thread). The form of a thread we are particularly interested in is a sequence of built-in body goals possibly followed by a tail-recursive call. The analysis can be done using an abstract domain $\{bound, unbound\}$ and proceeds as follows:

1. reorder body goals in each program clause using mode information to form a potential thread, and
2. perform abstract execution of the `main` program (or the top-level program of the program module being analyzed) according to the obtained control flow until the instantiation state of goal arguments reaches a fixpoint.

If the abstract interpretation guarantees the sequential execution of reordered goals to determine all the necessary values of input arguments, we have succeeded in forming a thread and can proceed to loop optimization described in the next section.

5.5 Loop Optimization

One of the most important applications of constraint-based analysis and abstract interpretation is the optimization of loops programmed as simple or mutual tail recursion.

Since tail-recursive calls usually have more statically available information about their arguments than the initial call to the predicate, it is reasonable to have two entry points for each predicate, one for external calls and the other for the tail-recursive loop. The purpose is to eliminate

1. tag operations and
2. general procedures for accessing generic objects

from the loop and to cache KL1 data using C variables while looping.

Abstract interpretation may not guarantee that all the data examined in clause guards in a loop have been instantiated and fully dereferenced when the loop is entered. In this case, a compiler may insert a synchronization code outside the loop to guarantee that all the data examined inside the loop have concrete values and thus

to reduce the number of synchronization operations performed at runtime. It is not always possible to move a synchronization point out of the loop because it may block the execution of unification body goals whose results are otherwise observable from other concurrent processes. However, in many cases we can prove that this won't block the publication of any output data.

5.6 Number Arrays

Loop optimization is effective for both scalar and vector computation. However, vectors in KLIC are not as efficient as they could be to represent homogeneous arrays of instantiated data (Section 5.2). To achieve performance competitive with programs written in procedural languages, we have implemented arrays of 64-bit floating point numbers and arrays of 32-bit integers as KLIC's data objects. They are supposed to be used with static analysis and have the following features:

1. All the elements of an array must be of the same type (64-bit floating-point numbers or 32-bit integers).
2. The values to be stored into arrays must be instantiated.
3. Linearity analysis must guarantee that multi-version control can be safely omitted.
4. Allocated on a special area not under management of the garbage collector. This is to avoid copying of large arrays by garbage collection. In an implementation on a shared-memory parallel computer we are currently working on, arrays may be allocated on shared memory.
5. No bound check of index values. It would be more reasonable to separate array bound checks from access operations so that static analysis may eliminate checks that are known to succeed.
6. Split and join operations with no copying. Suppose we want to let processes access and update different parts of an array concurrently and without interference. This can be done by splitting the original array and giving the resultant subarrays to the processes. The subarrays finally returned by the processes can be rejoined without copying, as long as they have been updated in place.

5.7 Experiments

To demonstrate the effect of our optimization techniques, we took two examples, one to compute $\sum_{k=1}^{10000}(1/k^2)$ and the other to compute the product of two 100×100 matrices, and compared the performance of

- the C code generated by KLIC (version 3.002),
- optimized, hand-compiled intermediate code (in C) we have designed, and
- programs directly written in C.

Table 5.1: Optimization of floating-point numbers and loops.

without opt.	with opt.	C
32.6 msec.	4.47 msec.	2.63 msec.
(12.4)	(1.70)	(1.00)

Table 5.2: Arrays vs. vectors.

KLIC vector	double array	double array with loop opt.	C
8.66 sec.	6.54 sec.	0.772 sec.	0.576 sec.
(15.0)	(11.4)	(1.34)	(1.00)

The results are shown in Tables 5.1 and 5.2. The measurements were done using Sun Ultra Enterprise 4000 (MPU: 168MHz), and the numbers shown are the execution times of the main loops. All the C programs were compiled using `gcc -O2 -fomit-frame-pointer`. The results show that loop optimization and array types were both effective, and the optimized KL1 programs turned out to be only 34%–70% slower than the comparable C programs. One of the remaining sources of overhead is the polling of external events in each iteration, without which a thread executing a tight loop might run indefinitely.

5.8 Conclusions

We have shown that static analysis can make the performance of KLIC, an implementation of a "pure" concurrent logic language, quite close to C for numerical computation. The most important future work is to build an optimizing KLIC compiler that makes use of the output of the static analyzer. Another important direction is to extend our array objects to allow parallel processing on shared-memory parallel computers.

A key feature of concurrent logic languages is (and should be) that parallelization can be achieved with very low additional programming effort. We hope our research will open up a new approach to high-performance computing and new application of concurrent logic programming.

第6章　並行論理プログラムの参照数解析

関数型言語や論理型言語をはじめとする記号処理言語は，自動記憶管理機構とポインタの隠蔽とによって，プログラミングの容易さとメモリ参照の安全性を実現している．現在もっとも広く使われている技法はコピーによるゴミ集め (copying garbage collection) であり，さまざまな望ましい性質をもっている．だが，もしプログラム実行時に起きる記憶域の割付けや参照に関して静的な（= コンパイル時の）解析ができれば，手続き型プログラムにより近い性能のコードを生成することができる．また並列分散処理や実時間処理のためには，ゴミ集めによらない記憶管理が可能なプログラムのクラスが同定できることが非常に重要である．並行論理型言語 Moded Flat GHC におけるそのような解析の基本的な技法として，参照数解析の枠組みを提案し，解析の安全性を証明する．参照数解析とは，複数の参照ポインタをもちうるデータ構造と，単一の参照ポインタしかもたないデータ構造とを，静的プログラム解析によって区別するものであり，後者に対してはコンパイル時ゴミ集めや記憶域の局所的再利用 (local reuse) が可能となる．提案する技法は，以前に提案したモード解析技法と同様，制約に基づく定式化を特徴としており，すでに，KL1 用静的解析系 klint 第 2 版の機能の一つとして部分的に実装されている．

6.1　はじめに

プログラム言語の変数は，記憶手段と見ることも通信手段と見ることもできる．
変数を通信手段として見ると，

- （ある場所，時刻に）そこへ値を代入することが送信に対応し，
- （他の場所，時刻に）その値を読み出すことが受信に対応する．

一度代入された値は，次の代入によって変更される前に[1]，1 回以上読み出されるのが普通である．ちょうど 1 回読み出される場合は 1 対 1 通信，2 回以上読み出される場合は 1 対多通信を行なっているとみなすことができる．

通信される内容がスカラー値の場合は話は簡単であるが，構造値（structured value）の場合は，それをヒープ上に作成して，ポインタの授受によって通信を行なうことが多い．このような構造値のための記憶域は，「最後の読み手」が処理系に返却したり，他目的に転用したりしてもかまわないが，これを実現するためには，個々の変数の読出し操作が，その変数の現在の値に関する「最後の読出し」であるか否かがわかっていないといけない．一般にはこの解析は困難であるので，ゴミ集め（garbage collection）によってヒープの記憶管理を行なうことが多い．

[1]単一代入変数の場合は，いったん決まった値は未来永劫変更されることはない．

（プログラム）	P ::= C の集合	(1)
（プログラム節）	C ::= A :-| B	(2)
（ボディ）	B ::= G のマルチ集合	(3)
（ゴール）	G ::= $T_1 = T_2$ | A	(4)
（非単一化ゴール）	A ::= $p(T_1,\ldots,T_n)$, $p \neq$ '='	(5)
（項）	T ::= （一階述語論理と同様）	(6)
（ゴール節）	Q ::= :- B	(7)

図 6.1: GHC のサブセットの構文

しかし，もし「ある変数は 1 対 1 通信にしか用いていない」ということが静的解析で保証できたら，その変数の値を読み出した時点で，変数値の占める記憶域を処理系に返却したり再利用したりすることができる．並行プログラミング言語では，1 対多通信の場合の「最後の読み手」を同定することは困難だが，1 対 1 通信ならば，唯一の読み手が最後の読み手であるから，話が簡単になる．

本論文では，単一代入変数（論理変数）を用いて通信を行なう並行論理型言語において，このような「1 対 1 通信と 1 対多通信の区別」，すなわち参照数解析を行なうための理論的枠組を提案する．具体的な言語としては，強モード体系をもつ Moded Flat GHC [46] を用いる．これは，データの流れが静的に推論できる言語の方が，静的な参照数解析が的確にできるからである．

これまでに書かれてきた並行論理プログラムを観察すると，プログラム中の大多数の変数は，1 対 1 通信に用いている [37]．特に，未完成メッセージやストリームのストリームなど，複雑なプロトコルをもつ通信は，ほぼすべてが 1 対 1 通信である．したがって，参照数解析は記憶管理の最適化に大変役立つと期待される．

6.2 並行論理プログラムの参照数

GHC[2]は，図 6.1 のような簡潔な構文をもつ並行論理型言語である（本論文では話を単純にするため，ガードゴールはないものとする）．

GHC プログラムの実行は，初期ゴール節から始まる，ゴール節の簡約（reduction）の繰返しである．ゴール節の簡約には

- 非単一化ゴールの（他ゴールへの）簡約（代入の観測＝受信）
- 単一化ゴールの簡約（代入の生成＝送信）

の 2 種類がある．ゴール節 G から G' への 1 ステップの簡約について簡単に復習しておく．

- **非単一化ゴール $g \in G$ を "h :-\| B" の形の節 $C \in \mathcal{P}$（新しい変数で名前替えしたもの）で簡約した場合．** GHC の同期の規則から，$g = h\theta$ であるような代入 θ があ

[2]並行論理プログラムの基本計算モデルについて議論するときは GHC と呼び，KL1 特有の機能について議論するときと区別することにする．

り，$G' = G\backslash\{g\} \cup B\theta$ となる．ここで '\' と '∪' それぞれ，マルチ集合の差と和をとる演算である．

- **単一化ゴール $(t_1 = t_2) \in G$ を簡約した場合．** t_1 と t_2 との単一化によって作られる代入（most general unifier）を θ とすると，$G' = (G\backslash(t_1 = t_2))\theta$ となる．単一化は一般には成功するとは限らないが，well-moded なプログラムでは，出検査（occur check）に失敗しない限り成功する [46].

どちらの場合も，簡約時には，一般に変数（v とする）への項（t とする）の代入操作が行なわれる．つまり，非単一化ゴールの簡約時には，プログラム節の中の局所変数（を renaming したもの）への代入が起き，単一化ゴールの簡約時には，ゴール節の中の変数への代入が起きる．ここで，それらの変数が複数の読み手出現を持つ場合，簡約後のゴール節の t への参照数が 1 以上増加する[3]．参照数解析とは，このような複数の読み手が参照する可能性のあるデータが，どの述語のどの引数がとりうるデータ構造のどの部分に出現しうるかを静的に解析するものである．

1 対多通信のための変数から指された経歴をもたない項は，複数参照となることがなく，不要になるタイミングがコンパイル時に把握できる．そこで，ガーベジコレクションによらずに記憶領域の回収や再利用を行なうことができる．

6.3 用語の定義

GHC プログラムの変数の出現のうち，節頭部における 2 回目以降の出現や，ガードゴールにおける出現を除く出現を，**チャネル出現**（channel occurrence）という．一般に GHC の変数は，1 回限り通信（one-shot communication），またはストリーム通信などの繰返し通信の通信路と考えることができるが，チャネル出現は通信路の端点とみなすことができる．

プログラム節やゴール節（初期ゴール節と，計算が進んだ後のゴール節の両方を含む）の中に 2 回以下しかチャネル出現しない変数を **linear な変数**と呼び，3 回以上チャネル出現している可能性のある変数を **nonlinear な変数**と呼ぶ．

Well-moded なプログラムでは，個々の変数がとりうるデータ構造の各部について，その値を決めることができる出現（書き手出現）はちょうど一つであることが保証されるので，ちょうど 2 回チャネル出現するということは，1 対 1 通信に利用しているということを意味している．また，1 回しかチャネル出現しない変数は 1 対 0 通信，つまりデータの廃棄を意味している．

また，$\langle symbol, arg\rangle$ の形の，関数/述語記号と引数位置との対を並べたものをパスと呼ぶ．P_{Atom} を，ゴールの中の記号の出現を指定するためのパスの集合とし，P_{Term} を，項の中の記号の出現を指定するためのパスの集合とする．たとえばゴール `p(f(a,b),C)` の `b` は，パス $\langle \mathtt{p}, 1\rangle\langle \mathtt{f}, 2\rangle \in P_{Atom}$ に出現する．

[3]本論文では，構造体の変数への代入は，複写による代入ではなく共有による代入によって行なうものと仮定する．

6.4 参照数標記

単一参照のデータ構造と複数参照のデータ構造を区別するために，ゴール節，およびプログラム節のボディゴール中のすべての関数記号 f に，f^1 または f^ω のように，1 または ω の標記をつけることを考える[4].

直感的には，ゴールの簡約時に起きる代入操作 $\{v \leftarrow t\}$ を，t へのポインタの代入によって実現した場合，複数のポインタから指される可能性のある構造体が標記 ω を持ち，単一のポインタからしか指されることのない構造体は標記 1 を持つ．1 ビットの参照カウンタ（reference counter）をモデル化したものと言うこともできる．

この標記は次のように管理される．

1. プログラム節および初期ゴール節の中の関数記号の標記は，それらが形成する構造体をどのように実現するかにしたがってつける．たとえば，

   ```
   :- p([1,2,3,4,5],X), q([1,2,3,4,5],Y).
   ```

   というゴール節があった場合，もし両ゴールの第 1 引数のリストを共有するならば，各関数記号に ω をつけなければならない．別々のリストを作成して渡すならば，標記はどちらでもよい．

   なお，標記 ω の主関数記号（principal function symbol）をもつ項の中に出現する関数記号は，すべて標記 ω をもっていなければならない（閉包条件）．だが，標記 1 の主関数記号をもつ項の中に出現する関数記号の標記は，必ずしも 1 であるとは限らない．
2. ゴールの簡約時に代入操作 $\{v \leftarrow t\}$ が起きるとする．
   (a) v が nonlinear な変数の場合，この代入操作は複数個の v の出現を t に具体化するので，t は複数参照となる（つまり，この代入操作を t へのポインタの代入によって実現すると，t へのポインタが複数本作られる）．そこで t の中に含まれるすべての構造体は複数参照に転じたと見なし，その構造体を表わす関数記号を ω に変更した上で，各 v の t への書換えを行なう．
   (b) v が linear な変数の場合は，代入操作 $\{v \leftarrow t\}$ はただ一つの v を具体化するだけなので，この代入操作によって参照数が増加することはない．そこで，t がもっている標記を変更せずに v の t への書換えを行なう．

6.5 参照数制約

Well-moded なプログラムの線形性を，関数 $\lambda : P_{Atom} \rightarrow \{nonshared, shared\}$ によって特徴づけることを考える．

その動機は，ω 標記の関数記号をもったデータ構造が現れるようなパスとそうでないパスを静的に区別することにある．「$\lambda(p) = nonshared$ であるようなパス p に，ω 標記の関数記号をもったデータ構造が現れることがない」ことが証明できれば，$nonshared$ なパスに現れるデータ構造の読み手は，読後のデータを廃棄することができる．

[4] この記法は，文献 [16, 31] のものを採用した．

(BF_λ) B 中のパス p に関数記号 f^ω が出現するとき，$\lambda(p) = shared$

(LV_λ) Linear な変数が p_1 と p_2 に出現するとき，

$\forall q \in P_{Term}(m(p_1q) = in \wedge \lambda(p_1q) = shared \Rightarrow \lambda(p_2q) = shared)$
（p_1 が頭部のパスのとき）

$\forall q \in P_{Term}(m(p_1q) = out \wedge \lambda(p_1q) = shared \Rightarrow \lambda(p_2q) = shared)$
（p_1 がボディゴールのパスのとき）

(NV_λ) Nonlinear な変数がパス p に出現するとき，

$\forall q \in P_{Term}(m(pq) = out \Rightarrow \lambda(pq) = shared)$　（p が頭部のパスのとき）

$\forall q \in P_{Term}(m(pq) = in \Rightarrow \lambda(pq) = shared)$
（p がボディゴールのパスのとき）

(BU_λ) $=_k$ をボディの単一化ゴールとすると，

$\forall q \in P_{Term}(\langle =_k, 1\rangle q = shared \Leftrightarrow \langle =_k, 2\rangle q = shared)$

図 6.2: 節 h :-| B が課する参照数制約

プログラム節 h :-| B またはゴール節 :- B が，参照数を表わす関数 λ に課する制約（参照数制約）は，図 6.2 の通りとする．この参照数制約は，プログラムのモード（関数 m で表現）を参照している．モード制約規則 [46] は図 6.3 に示しておく．m を，プログラム全体のモードを表わす関数とするとき，m をパス p の位置から眺めたサブモード m/p を，$(m/p)(q) = m(pq)$ を満たす関数として定義する．また，IN および OUT を，それぞれ，常に in および out を返すようなサブモードと定義する．上線 '¯' は（モード，サブモード，またはモード値の）極性を反転する記号である．

モード制約や参照数制約の規則で，単一化ゴールに番号をふっているのは，異なる単一化ゴールが異なるモードや参照数をもつこと（多相性）を許すためである．

図 6.2 の参照数制約は，すべてのパス p について $\lambda(p) = shared$ とすれば自明に満たされる．しかしそれでは参照数解析の目的は果たされない．参照数解析の目的は，上記の制約を満たす最小の λ を求めることである．ここで「最小」とは，二つの参照数関数 λ_1, λ_2 の間の半順序関係を

$$\lambda_1 \leq \lambda_2 \quad \Leftrightarrow \quad \forall p \in P_{Atom}(\lambda_1(p) = shared \Rightarrow \lambda_2(p) = shared)$$

と定めたときの最小性を意味する．（最小解が存在するかどうかは証明しなければならないことだが，ここではふれないことにする．）

6.6 主部簡約定理

下記の基本定理が成り立つ．

定理 1　（主部簡約，subject reduction）λ が，プログラム $\mathcal{P}$ とゴール節 G の参照数制約を満たすとする．G が 1 ステップ簡約されて，ゴール節 G' になったとすると，λ は G' の

(HF) h 中のパス p に関数記号が現れるならば $m(p) = in$

(HV) h 中のパス p に，h 中に複数回出現する変数が現れるならば $m/p = IN$

(BU) $=_k$ をボディの単一化ゴールとすると $m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}$

(BF) ボディ・ゴールの中のパス p に関数記号が現れるならば $m(p) = in$

(BV) 変数 v が h および B 中に $n\,(\geq 1)$ 回，$p_1, \ldots, p_n$ に出現し，うち h 中の出現は $p_1, \ldots, p_k$ $(k \geq 0)$ であるとする．このとき $\mathcal{R}(S)$ を，どのパス q についても $\exists s \in S(s(q) = out \wedge \forall s' \in S \backslash \{s\}\,(s'(q) = in))$ を満たす述語とすると，

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & k = 0 \text{ の場合;} \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & k > 0 \text{ の場合} \end{cases}$$

図 6.3: 節 h :-| B が課するモード制約

参照数制約も満たす．ただし，ここで実行されたゴール $g \in G$ は，拡張出現検査 (extended occur check) に失敗する単一化ゴール（つまり，同一変数どうし，または変数とその変数を含む項とを単一化しようとするゴール）ではないとする．

証明　場合分けが非常に多いので，いくつかの主要な場合について詳述し，残りは概略を述べる．

簡約の種類によって二つの場合に分ける．記法を簡略にするため，ここではゴール節と，そのゴール節がもつボディゴールのマルチ集合とを同一視する．

[場合 1] この簡約で，非単一化ゴール g を，"h :-| B" の形の節 $C \in \mathcal{P}$（新しい変数で名前替えしたもの）で書き換えた場合．

我々が検討しなければならないのは，$g \in G$ に出現する変数が課する制約 (LV_λ), (NV_λ) と，θ によって $B\theta$ $(\in G')$ に持ち込まれた関数の出現（これらの出現は，この関数の g における出現に端を発するものである）が課する制約 (BF_λ) である．G' 中の他の変数が課する制約と，G' 中の関数の他の出現（それらはすでに，$G \backslash \{g\}$ か B の中にあったものである）が課する制約は，簡約前とまったく同じであるから考えなくてよい．そこで，g 中に現れる各記号について考える．

1. パス p に関数記号 f^κ $(\kappa \in \{1, \omega\})$ が出現する場合．この場合は，
   (a) H 中のパス p の記号が f^κ であるか，さもなければ
   (b) p' を p のプレフィクスとする（つまり $p = p'q$，$p' \in P_{Atom}$, $q \in P_{Term}$ なる q が存在するとする）と，H 中のパス p' に変数（v とする）が出現する．

   1a の場合は，この出現は簡約時に消えるので，制約 (BF_λ) は適用されない．そこで 1b の場合だけ考えればよい．1b では，f^κ の出現が $B\theta$ $(\in G')$ に新たに持ち込まれる可能性がある．v が，B 中に $n\,(\geq 0)$ 回，$r_1, \ldots, r_n$ に出現するとし，g_j を，r_j における出現が属するゴールとしよう．ゴール $g_i\theta$ のパス $r_i q$ に記号 f^κ が出現するこ

とになる．

(a) $n \leq 1$（vが linear な変数）の場合：$n = 1$ の場合のみ考えれば十分である。もし $\kappa = \omega$ ならば，$\lambda(r_i q) = shared$ でなければならない．ところが

i. $\kappa = \omega$ ならば，G が (BF_λ) を満たすという仮定から $\lambda(p) = shared$

ii. モード制約規則 (BF) から $m(p) = in$

iii. (LV_λ) と 1(a)ii から，$\lambda(p) = shared \Rightarrow \lambda(r_i q) = shared$

iv. 1(a)i と 1(a)iii から $\lambda(r_i q) = shared$

(b) $n > 1$（vが nonlinear な変数）の場合：第 6.4 節の規則 2a から，$g_i\theta$ のパス $r_i q$ の記号は，f^ω となる．一方，

i. モード制約規則 (BF) から $m(p) = in$

ii. 1(b)i とモード制約規則 (BV) から $m(r_i q) = in$

iii. 1(b)ii と (NV_λ) から $\lambda(r_i q) = shared$

よって，いずれの場合も，G' は (BF_λ) を満たす．

2. パス p に変数 w が出現する場合．w は g に l (≥ 1) 回，$p_1(= p), p_2, \ldots, p_l$ に出現し，$G\backslash\{g\}$ には n (≥ 0) 回，$p_{l+1}, \ldots, p_{l+n}$ に出現するとする．
$g = h\theta$ であるような代入 θ があるので，各 p_i $(1 \leq i \leq l)$ について，H 中のパス p'_i が変数（v_i とする）であるような p_i のプレフィクス $p'_i \in P_{Atom}$ と，$p_i = p'_i q_i$ をみたす $q_i \in P_{Term}$ が存在する．v_i は B に n_i (≥ 0) 回，$r_{i1}, \ldots, r_{in_i}$ に出現するとする．
すると w は $B\theta$ 中の，$r_{ij}q_i$ $(1 \leq i \leq l, 1 \leq j \leq n_i)$ に現れることになる[5]．
さて，ある v_i が nonlinear な変数の場合，G' 中の w は nonlinear な変数になる可能性がある．w は，

- $r_{ij}q_i$ $(1 \leq i \leq l, 1 \leq j \leq n_i)$（$B\theta$ による持込み）
- $p_{l+1}, \ldots, p_{l+n}$（$G\backslash\{g\}$ からの引継ぎ）

に出現するので，これらのパスをプレフィクスにもつパスについて検討する．
まず前者については，

- 節 C が (NV_λ) を満たすことから $m(r_{ij}q_i q) = in \Rightarrow \lambda(r_{ij}q_i q) = shared$

したがって，$B\theta$ によって持ち込まれた w の出現パスに関しては (NV_λ) は満たされる．
次に後者については

(a) $m(pq) = in$ の場合

i. 仮定とモード制約 (BV) から $\exists k \leq n\,(m(p_{l+k}q) = out)$

ii. 2(a)i から $\exists k \leq n\,(m(p_{l+k}q) = in) \Rightarrow n > 1$，つまり $\exists k \leq n\,(m(p_{l+k}q) = in)$ ならば，変数 w は G 中で nonlinear

iii. 2(a)ii と (NV_λ) から $\forall k \leq n\,(m(p_{l+k}q) = in \Rightarrow \lambda(p_{l+k}q) = shared)$

(b) $m(pq) = out$ かつ w が G 中で linear な変数の場合（この場合 $n \leq 1$ である）

i. 節 C が (NV_λ) を満たすことから

$$\forall q \in P_{Term}(m(pq) = out \Rightarrow \lambda(pq) = shared$$

[5] w の出現回数は $n_1 + \cdots + n_l$ より少ないかも知れない．g 中の w の二つの出現を，h 中の同じ変数の異なる出現が受け取ると，それらは $B\theta$ に独立には持ち込まれないからである．だが，このことは以下の議論には影響しない．

ii. 2(b)i と仮定から $\lambda(pq) = shared$

iii. 2(b)ii と仮定と (LV_λ) から $\forall k \leq n\,(\lambda(p_{l+k}q) = shared)$

(c) $m(pq) = out$ かつ w が G 中で nonlinear な変数の場合

i. 節 C が (NV_λ) を満たすことから

$$\forall k \leq n\,(m(p_{l+k}q) = in \Rightarrow \lambda(p_{l+k}q) = shared)$$

よって，$G\backslash\{g\}$ から引き継いだ w の出現パスについても (NV_λ) は満たされる．すべての v_i が linear な変数の場合は，w が G 中で linear ならば w は G' 中でも linear であり，G 中で nonlinear ならば G' 中でも nonlinear である．それぞれの場合について，上と同様に確かめることができる．

[場合 2] この簡約では，単一化ゴール $t_1 =_k t_2$ を実行した．well-moded であるという仮定から，$m(\langle =_k, i\rangle) = out$ であるような i が存在する．一般性を失うことなく，$i = 1$ である（つまり，単一化は常に，左辺の変数への代入である）と仮定してよい．拡張出現検査の仮定により，項 t_2 は，変数 t_1 自身や，それを含む項ではない．したがって $\theta = \{t_1 \leftarrow t_2\}$ とすると，$G' = (G\backslash\{t_1 =_k t_2\})\theta$ である．変数 t_1 が，G 中に他に $n\ (\geq 0)$ 回，$r_1, \ldots, r_n$ に出現すると仮定しよう．このとき，G' には，t_2 中の各記号が n 回ずつ複写されて出現する．そこでこれらの記号が参照数制約 (BF_λ)，(LV_λ)，(NV_λ) を満たすかどうかを検討すればよい．

1. t_2 のパス q に関数記号 f^κ が出現する場合．制約 (BF_λ) の内容から，$\kappa = \omega$ の場合のみ考えればよい．
   (a) (BF_λ) から $\lambda(\langle =_k, 2\rangle q) = shared$
   (b) 1a と (BU_λ) から $\lambda(\langle =_k, 1\rangle q) = shared$
   (c) モード制約 (BF) から $m(\langle =_k, 2\rangle q) = in$
   (d) 1c とモード制約 (BU) から $m(\langle =_k, 1\rangle q) = out$
   (e) 1d とモード制約 (BV) から $m(r_i q) = in (1 \leq i \leq n)$
   (f) t_1 が linear のときは，1b，1d と (LV_λ) から，$\lambda(r_1 q) = shared$
   (g) t_1 が nonlinear のときは，1e と (NV_λ) から，$\lambda(r_i q) = shared (1 \leq i \leq n)$
   ゴール $t_1 =_k t_2$ の実行により，f^ω は G' 中の $r_1 q, \ldots, r_n q$ に新たに出現するようになる．しかし上に示すように，λ はこれらの出現が課する制約 (BF_λ) を満足している．
2. t_2 のパス q に変数 $w\ (\neq t_1)$ が出現する場合．w はゴール $t_1 =_k t_2$ 中に $l\ (\geq 1)$ 回，$p_1 (= \langle =_k, 2\rangle q), p_2, \ldots, p_l$ に出現し，$G\backslash\{t_1 =_k t_2\}$ には $m\ (\geq 0)$ 回，$p_{l+1}, \ldots, p_{l+m}$ に出現するとする．ゴール $t_1 =_k t_2$ の実行により，w は新たに $r_1 q, \ldots, r_n q$ および $r_1 p_i, \ldots, r_n p_i\ (1 \leq i \leq l)$ に出現する．これらのパスの参照数制約を調べればよい．t_1 が G 中で nonlinear な変数の場合は，G' の w も nonlinear になる可能性がある．ところが t_1 に対する制約 (NV_λ) から，各 $j\ (1 \leq j \leq n)$ について $\forall s \in P_{Term}(m(r_j s) = in \Rightarrow \lambda(r_j s) = shared$ である．したがって G' 中の w の新たな出現は，すべて (NV_λ) をみたす．
   t_1 が G 中で linear な変数の場合は，w が G 中で linear ならば G' でも linear である．$l = 1$，すなわち G 中の w のもう一つの出現が，やはり t_2 中にある場合のみを考え

れば十分である．パス s について $m(\langle =_k, 2\rangle qs) = out \wedge \lambda(\langle =_k, 2\rangle qs) = shared$ と仮定すると，

(a) モード制約 (BV) と (LV_λ) から

$$m(\langle =_k, 2\rangle p_2 s) = in \wedge \lambda(\langle =_k, 2\rangle p_2 s) = shared)$$

(b) 2a とモード制約 (BU) と (BU_λ) から

$$m(\langle =_k, 1\rangle p_2 s) = out \wedge \lambda(\langle =_k, 1\rangle p_2 s) = shared)$$

(c) 2b と (LV_λ) から $\lambda(r_1 p_2 s) = shared$

ところがこれは，G' 中の変数 w が課する制約 (LV_λ) の後件にほかならない．

t_1 が G 中で linear，w が G 中で nonlinear の場合は，w は G' 中でも（一般に）nonlinear である．この場合もほとんど同様にして，G 中の参照数制約から，G' 中の変数 w が課する制約 (NV_λ) を導くことができる．（証明終）

6.7 参照数解析の応用

参照数解析には，つぎのような応用がある．

1. **データ構造の局所的再利用** — データ構造の唯一の読み手は，読み終えた部分の記憶域を，新たなデータ構造の生成のために転用することができる．これによって，破壊的代入の概念をもたない言語において，処理系のレベルではデータ構造の update in place が可能になる．Lisp の `nconc` や `rplacd` のような機能をプログラマに見せる必要もなくなる．
   宣言型言語で配列処理を行なう場合，配列が複数参照となりうるならば，(コピーは禁止的に高価なので）多版（multi-version）データ構造のような実装法をとらざるを得ない．しかし，単一参照性が保証できれば，多版構造を作る必要がなくなる．したがって，静的参照数解析は，宣言型言語における高効率の配列処理のために本質的に重要である [39].
   参照数解析を行なうと，一つの配列を in-place で分割併合することが可能になり，配列の異なる部分を並列に更新するような処理も可能になる．
2. **分散処理** — サイトをまたがるポインタを単一参照のものに限定することができるような分散処理応用では，大域的なゴミ集めが不要になり，ポインタの輸出入管理も大幅に単純化される．したがって，宣言型言語のネットワークプログラミングへの適用が，より現実的なものとなる．
3. **実時間処理** — ロボットの制御など，(弱い意味での）処理の実時間性を要するプログラムでは，ゴミ集めによる処理の中断が問題となる．コンパイル時ゴミ集めはこの問題に対する一つの解決策となる．

また，参照数解析は，プログラムの実行に必要な記憶容量の解析にも有効であると期待できる．

```
:- module main.
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).

qsort([],    Ys0,Ys ) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
   part(X,Xs,S,L), qsort(S,Ys0,Ys1), Ys1=[X|Ys2], qsort(L,Ys2,Ys3).

part(_,[],    S, L ) :- true | S=[], L=[].
part(A,[X|Xs],S0,L ) :- A>=X | S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0=[X|L], part(A,Xs,S,L).
```

図 6.4: クイックソートプログラム

6.8 実装—klint 第 2 版

KL1 プログラム静的解析系 klint 第 2 版 [39] は，モード解析に加えて参照数解析と型解析機能をもっている．

klint 第 2 版の参照数解析では，制約 (LV_λ) のかわりに，より簡単で強力な制約

- (LV'_λ) $\forall q \in P_{Term}(\lambda(p_1 q) = shared \Leftrightarrow \lambda(p_2 q) = shared)$

を採用している．klint 第 2 版では，参照数制約を，参照数グラフと呼ばれる素性グラフ (feature graph) を作ることによって解いているが，この近似により，linear な変数に関する参照数制約が，グラフの節点どうしの単一化によって解けるようになるからである．

明らかに，(LV'_λ) $\Rightarrow$ (LV_λ) が成り立つ．この近似を行なっても，大多数のプログラムでは，単一参照のパスの検出に支障は生じない．しかし，配列を数表として用いる場合のように，初期化時には単一参照性が保たれるものの，その後多重参照に転じるようなデータ構造については，上記の近似を行なうと，初期化時から多重参照であると判断されてしまう．

例として，図 6.4 のクイックソートプログラムを，klint にかけた結果を以下に示す．

```
*** Linearity Graph ***
node(0): (unconstrained)
 <(main:qsort)/2,1> ---> node(24)
 <(main:qsort)/2,2> ---> node(16)
 <(main:qsort)/3,1> ---> node(24)
 <(main:qsort)/3,2> ---> node(16)
 <(main:qsort)/3,3> ---> node(16)
 <(main:part)/4,1> ---> SHARED
 <(main:part)/4,2> ---> node(24)
 <(main:part)/4,3> ---> node(24)
 <(main:part)/4,4> ---> node(24)
node(24): (unconstrained)
```

```
 <cons,2> ---> node(24)
node(16): (unconstrained)
 <cons,1> ---> SHARED
 <cons,2> ---> node(16)
```

これは，関数λの値を参照数グラフとして表示したものである．SHAREDと書かれたパス，すなわち

- partの第1引数，
- 2引数のqsortの第2引数のリストの各要素，および
- 3引数のqsortの第2，3引数のリストの各要素

は，2引数のqsortの第1引数からの入力リストが単一参照か複数参照かにかかわらず，複数参照になる．しかしこれらのパスは，同時に行なわれる型解析によって整数型，すなわちスカラーであることがわかるので問題ない．一方，このプログラムが生成するリストの骨格は，少なくとも入力リストが単一参照である限り，単一参照であることが保証される．

6.9 関連研究

並行論理型言語Flat GHCの処理系については，MRB（multiple reference bit）と呼ばれる1ビット参照カウンタを動的に管理する方式が提案され[6]，並列推論マシン上のKL1処理系にも実装された[21]．本論文で提案する参照数解析は，大まかに言えば，この1ビット参照カウンタの値を静的に解析することにより，参照カウンタの保持や管理を不要にする技法であると言える．

文献[21]には，ポインタの輸出入表や重みつき輸出カウント（weighted export counting）をはじめとする並行論理型言語の分散実装のさまざまな技法が提案されているが，参照数解析が可能なデータは，管理を大幅に単純化することができる．

文献[16]では，並行計算の理論的な枠組みであるπ計算（pi-calculus）に，参照数情報を保持した型概念を導入し，その解析技法を提案している．Moded Flat GHCのモード概念や参照数概念は，本質的には型概念の拡張にほかならない．

関数型言語の枠組では，文献[31]で，やはり型に参照数概念を導入して，その解析技法を提案している．構造体に標記をつける点は本論文の技法と類似であるが，関数型言語における参照数解析は評価規則のバラエティや高階関数の処理などに難しさがあり，並行論理型言語における参照数解析は複雑な情報の流れを扱うところに難しさがある．

6.10 結論と今後の課題

並行論理型言語Moded Flat GHCの参照数解析のための体系を提案し，主部簡約定理を示した．参照数解析をモード解析や型解析と併用することにより，並行論理プログラムから，より手続き型言語に近いコードを生成できるようになる．また，並列分散環境

のように，プログラムのコンパイルがより難しい実行環境で走る本格的なプログラムを，並行論理型言語のように単純な基本操作しかもたない並行言語で見通し良く記述し，高度な静的解析によって分散実行させる可能性が，現実のものとなってきた．今後は，これらの静的解析情報を活かして，並行論理型言語の高性能計算や分散計算への適用を図ってゆきたい．

参考文献

[1] Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.

[2] Ajiro, Y., Ueda, K. and Cho, K., Error-Correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP'98)*, LNCS 1520, Springer, 1998, pp. 40–54.

[3] Bowen, D. L. (ed.), Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D., *DECsystem-10 Prolog User's Manual.* Dept. of Artificial Intelligence, Univ. of Edinburgh, 1983.

[4] Chen, T. Y., Lassez, J.-L. and Port, G. S., Maximal Unifiable Subsets and Minimal Non-unifiable Subsets. *New Generation Computing*, Vol. 4 (1986), pp. 133–152.

[5] Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. PLILP'94*, LNCS 844, Springer, 1994, pp. 25–39.

[6] Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Logic Programming: Proc. of the Fourth Int. Conf (ICLP'87)*, The MIT Press, 1987, pp. 276–293.

[7] Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs. In *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96)*, The MIT Press, 1996, pp. 215–229.

[8] Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[9] Colmerauer, A., Prolog and Infinite Trees. In *Logic Programming*, Clark, K. L. and Tärnlund, S. -A. (eds.), Academic Press, London, 1982, pp. 231–251.

[10] Cohen, S., Multi-Version Structures in Prolog. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, pp. 265–274.

[11] Debray, S. A., Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Trans. Prog. Lang. Syst.*, Vol. 11, No. 3 (1989), pp. 418–450.

[12] Eriksson, L.-H. and Rayner, M., Incorporating Mutable Arrays into Logic Programming. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, 1984, pp. 101–114.

[13] Foster, I. and Taylor, S., Strand: A Practical Parallel Programming Language. In *Proc. 1989 North American Conf. on Logic Programming*, Lusk, E. L. and Overbeek, R. A. (eds.), MIT Press, Cambridge, MA, 1989, pp. 497–512.

[14] Hirata, M., Programming Language Doc and Its Self-Description or, $X = X$ is Considered Harmful. In *Proc. 3rd Conf. of Japan Society of Software Science and Technology*, 1986, pp. 69–72.

[15] Jaffar, J., Efficient Unification over Infinite Terms. *New Generation Computing*, Vol. 2, No. 3 (1984), pp. 207–219.

[16] Kobayashi, N., Pierce, B. C. and Turner, D. N., Linearity and the Pi-calculus. In *Proc. 23rd ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL'96)*, ACM, pp. 358–371 (1996).

[17] Lloyd, J. W., *Foundations of Logic Programming* (Second ed.). Springer-Verlag, Berlin, 1987.

[18] Pereira, F. C. N., Grammars and Logics of Partial Information. In *Proc. Fourth Int. Conf. on Logic Programming*, Lassez, J. -L. (ed.), The MIT Press, 1987, pp. 989–1013.

[19] ICOT PIMOS 開発グループ編，PIMOS マニュアル (第 3.0 版)—プログラミング編—，ICOT，1991 年 11 月．

[20] Milner, R., A Theory of Type Polymorphism in Programming. *J. of Computer and System Sciences*, Vol. 17, No. 3 (1978), pp. 348–375.

[21] Nakajima, K., Inamura, U., Ichiyoshi, N., Rokusawa, K. and Chikayama, T., Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. Sixth Int. Conf. on Logic Programming*, The MIT Press, pp. 436–451 (1989).

[22] Plotkin, G. D., A Structural Approach to Operational Semantics. DAIMI FN-19, Computer Science Dept., Aarhus Univ., Denmark, 1981.

[23] Reiter, R., A Theory of Diagnosis from First Principles. *Artificial Intelligence*, Vol. 32 (1987), pp. 57–95.

[24] Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM, 1990, pp. 232–245.

[25] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic Programming*, Debray, S. and Hermenegildo, M. (eds.), The MIT Press, 1990, pp. 431–446.

[26] Shapiro, E. Y. (ed.), *Concurrent Prolog: Collected Papers*, Vol. 1–2. The MIT Press, Cambridge, MA, 1987.

[27] Shapiro, E., The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

[28] Somogyi, Z., A Parallel Logic Programming System Based on Strong and Precise Modes. Ph. D. thesis, Tech. Report 89/4, Dept. of Computer Science, Univ. of Melbourne, Melbourne, Australia, 1989.

[29] Somogyi, Z., Henderson, F. and Conway, T., Mercury: An Efficient Purely Declarative Logic Programming Language. In *Proc. Australian Computer Science Conference*, Glenelg, Australia, 1995, pp. 499–512.

[30] Tick, E. and Koshimura, M., Static Mode Analyses of Concurrent Logic Languages, ICOT Tech. Report TR-875, ICOT, Tokyo, 1994. Also to appear in *J. Programming Languages Design and Implementation.*

[31] Turner, D. N., Wadler, P. and Mossin, C., Once Upon a Type. In *Proc. Seventh Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, ACM, pp. 1–11 (1995).

[32] Ueda, K., *Guarded Horn Clauses.* Doctoral Thesis, Faculty of Engineering, Univ. of Tokyo, 1986.

[33] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Tech. Report TR-208, 1986, ICOT. Also in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, Amsterdam, 1988, pp. 441–456.

[34] Ueda, K., Parallelism in Logic Programming. In *Information Processing 89*, Ritter, G. X. (ed.), North-Holland, Amsterdam, 1989, pp. 957–964.

[35] Ueda, K., Designing a Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, 1990, pp. 87–94.

[36] Ueda, K., I/O Mode Analysis in Concurrent Logic Programming. In *Proc. Int. Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, Ito, T. and Yonezawa, A. (eds.), LNCS 907, Springer, 1995, pp. 356–368.

[37] Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*, LNCS 1068, Springer, 1996, pp. 134–153.

[38] Ueda, K. and Cho, K. *kima* — Analyzer of Ill-moded KL1 Programs. Available from `http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H8-index-E.html`, 1997.

[39] Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H9-index-E.html`, 1998.

[40] Ueda, K., Linearity Analysis of Concurrent Logic Programs. Presented at the 11th Annual Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP'98), IPSJ SIGPRO, August 1998. Full paper in preparation.

[41] Ueda, K. and Chikayama, T., Efficient Stream/Array Processing in Logic Programming Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, pp. 317–326.

[42] Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.

[43] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988, pp. 582–591.

[44] Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, Warren, D. H. D. and Szeredi, P. (eds.), The MIT Press, 1990, pp. 3–17.

[45] Ueda, K. and Morita, M., Message-Oriented Parallel Implementation of Moded Flat GHC. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, Tokyo, 1992, pp. 799–808. Revised version in *New Generation Computing*, Vol. 11, No. 3–4 (1993), pp. 323–341.

[46] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.

[47] Wand, M., Finding the Source of Type Errors. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, ACM, 1986, pp. 38–43.

[48] Yoshida, K. and Chikayama, T., $\mathcal{A}'\mathcal{UM}$ — A Stream-Based Concurrent Object-Oriented Language. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988, pp. 638–649. Also in *New Generation Computing*, Vol. 7, No. 2–3 (1990), pp. 127–157.