Kima: 並行論理プログラム自動修正系

網代 育大　上田 和紀

強モード / 型体系の下で，並行論理プログラムの簡単な誤りを自動的に修正するシステム Kima の実装を行なった．Kima は KL1 プログラムにおける変数の少数個の書き誤りを，静的な解析によって自動的に修正することができる．並行論理プログラミングにおける強モード体系および強型体系は，通信のプロトコルやデータ型の一貫性を保証し，多くの誤りを静的に検出することを可能にする．このとき用いられるモード / 型解析は，多数の簡単なモード / 型制約の制約充足問題であり，プログラム中の誤りは制約集合の矛盾を引き起こすことが多い．そして制約集合の中から矛盾する極小の部分集合を求めることで，誤りの箇所を局所的に特定することができる．Kima は特定した場所の周辺の記号の書換えとモード / 型の再計算を機械的に行なうことによって自動修正を実現している．本論文では，自動修正のアルゴリズムや効率化手法について述べるとともに，いくつかの定量的な実験を行なって，その有効性について論じる．

1 はじめに

本研究では，プログラムの性質を静的に解析できる体系の下で，ユーザによってプログラムに関する仕様や宣言を与えることなく，プログラムの自動修正がどの程度可能かについて，枠組を与えるとともに実装を行ない，その有用性の検証を行なった．

強い型体系 (strong-typing) や強いモード体系 (strong-moding) を備えた言語では，型やモードを静的に検査ないしは推論することによって，それらに関するプログラムの誤りを事前に検出することが可能である．ML などの関数型言語の型解析 [10]は，プログラムテキストから得られる型制約式を制約充足問題 (constraint satisfaction problem) として解くものである．この場合の制約充足問題は単一化問題として効率よく解くことができる．

同様に，並行論理型言語における強モード体系は，プログラムテキストから得られるモード制約式を制約充足問題として解くことで，ユーザによってモード宣言やプログラムの仕様を与えることなく，プロセス間通信プロトコルの整合性 (ないしは変数に対する読み書きのケイパビリティ) を静的に判定することを可能にする [14]．このモード解析も，大部分は素性グラフ (feature graphs) の単一化問題 (unification problem) に帰着でき，プログラムサイズに関して，ほぼ線形オーダーで解くことができる [1]．並行論理型言語においては，データ型についても同様の解析が可能である．

プログラムの詳細な性質の解析には抽象解釈が広く用いられるが，より基本的な性質の解析について，制約充足の枠組で定式化することには多くの利点がある．たとえば，並行論理プログラムに誤りが存在する場合，そこから得られるモード制約の集合が充足不可能になることが極めて多く，その制約集合から矛盾する極小部分集合

Kima — an Automated Error Correction System for Concurrent Logic Programs.

Yasuhiro Ajiro, 早稲田大学大学院理工学研究科, Graduate School of Science and Engineering, Waseda University.
Kazunori Ueda, 早稲田大学理工学部, School of Science and Engineering, Waseda University.

コンピュータソフトウェア, Vol.18, No.0(2001), pp.122–137.
[論文] 2000 年 1 月 23 日受付.

を探索することによって，プログラムの誤りを局所的に特定できることがわかっている [5]．この技法を用いると，１回の解析で複数の独立な誤りを検出することもできる．

そして特定した節および変数記号を元に，その周辺の変数記号の網羅的な書換えおよびそのモード / 型の検査を機械的に行なうことでプログラムを自動的に修正することができる [2] [3]．これは generate-and-test 方式に他ならないが，モード / 型制約の矛盾する極小部分集合を求めて，誤りの可能性がある箇所をあらかじめプログラム中の一定の範囲内に絞り込むことにより，探索空間及び計算時間の増加を大幅に抑えることができる．この枠組に関して言えば，並行論理プログラミングにおいては，モード情報がデータ型情報よりも基本的な役割を果たすが，型情報の併用は探索空間の縮小や検出力の強化，修正案の品質の向上に非常に有効に作用する．

提案する技法は，あるモジュールに含まれる述語など，プログラムの一部に対しても適用できるという特徴を持つ．たとえば，プログラムが部分的にしか完成していない状況でも，本技法は十分な有用性を発揮する．これは，プログラムテキストから得られるモードや型に関する制約の集合が一般に冗長性を持つためである．現実にある多くのプログラムでは，

- 条件分岐や非決定的選択 (nondeterministic choices) が，並行論理型言語では複数の節によって表され，
- また，１つの述語が異なる複数の場所から呼ばれることが多い

ため，１つの述語に複数の同じ制約が課せられる．これが冗長性の存在する理由である．誤りの箇所を特定する際，この冗長性が高い場合は，矛盾の原因となっている制約をさらに正確に特定することもできる [7].

本研究では，対象言語として高速な実装が入手可能な並行論理型言語 KL1 [4] を採用し，静的解析の枠組として Moded Flat GHC [14]の備えるモード / 型体系を利用した．両言語はいずれも GHC をベースにしており，Moded Flat GHC のモード / 型体系は KL1 に対しても適用可能である．これらを組み合わせることによって，KL1 プログラムの軽微な誤りを自動的に修正するシステム Kima を開発した．Kima を用いた実験の結果，強モード / 型体系を仮定した KL1 プログラムの自動修正に関して有望な結果を得ることができた．

第２節では，自動修正技術の基礎となっている並行論理型言語における強モード / 型体系について概説し，第３節で静的解析による誤り箇所の同定技法について簡単に説明する．そして，第 4, 5 節を使って，本研究の中心である自動修正技術とその効率の良い実現法について解説した後，第 6, 7 節で実験結果と例を示す．

本論文は [2] [3] を基に，検出力の強化や探索の最適化などに関する説明の追加および実験結果の大幅な拡充を行なって作成したものである．

2 並行論理プログラミングにおける強モード / 型体系

ここでは自動修正技術の基礎となっている Moded Flat GHC の強モード / 型体系とモード / 型解析について簡単に説明する．詳細については [14] [15]を参照してほしい．

Moded Flat GHC のモード体系の目的は，ゴールのふるまいを定義する述語の引数に，極性構造 (データ構造の各部の情報の流れの向きを定めたもの) を与えることである．この極性構造は，データ構造のどの部分も，協調的に，ちょうど１つのゴールによって具体化されるように，モード解析によって計算される．

本モード体系におけるモードは，データ構造の各部を指定するための**パス**の集合から，集合 $\{in, out\}$ への関数である．パスとは，$\langle symbol, arg\rangle$ の形の，関数 / 述語記号と引数位置との対を並べたものであり，項 $Term$ と原子論理式 (ゴールや節頭部) $Atom$ のパスの集合は，依存型を用いて次のように定義される:

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

ここで，$Fun/Pred$ は関数 / 述語記号の集合，N_f/N_p は記号 f/p の引数位置を表わす番号の集合である．また，Σ は以下のような集合演算子を意味している:

$$\sum_{f\in Fun} N_f = \{\langle f, i\rangle \mid f \in Fun,\ i \in N_f\}$$

モード解析の目的は，すべての通信が協調的に行なわれるようなモードづけ $m : P_{Atom} \rightarrow \{in, out\}$ を求めることである．このようなモード m をプログラムの

(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 $m/p = IN$
(GV) 同じ変数が h 中の p と G 中の p' に出現するならばどのパス q についても $m(p'q) = in \Rightarrow m(pq) = in$
(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 $\exists s \in S\big(s(q) = out \wedge \forall s' \in S\backslash\{s\}\,(s'(q) = in)\big)$ を満たす述語とすると，

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

図 1 節 h :- G | B が課するモード制約

well-moding と呼び，またこのようにモードづけできるプログラムを well-moded なプログラムという．

m はプログラム全体のモードを表わすが，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'$\overline{\ \ }$'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．

プログラムは，h :- G | B (h は原子論理式，G および B は原子論理式のマルチ集合) の形の節の集合であるが，これが課する制約は，図 1 のようにまとめることができる．規則 (BU) 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を持つことを許すためである．図 1 のモード規則はどれも，すべての通信が協調的であるという仮定から導かれる．

例として，append プログラムを考える．

```
1: R1 : append([],    Y,Z ) :- true | Y=1Z.
2: R2 : append([A|X],Y,Z0) :- true |
3:            Z0=2[A|Z], append(X,Y,Z).
```

たとえば節 R_2 からは，次の 8 個の制約が得られる．ここで，"." は非空リストの主関数記号を表す．また a は述語記号 append の略記である．

R_1 からも 4 個の制約が得られるが，$=_k$ に関する制約を簡約化すると，次の 6 個の制約のみが残る:

モード制約	規則	原因
$m(\langle \mathsf{a}, 1\rangle) = in$	(HF)	"."
$m/\langle =_2, 1\rangle = \overline{m/\langle =_2, 2\rangle}$	(BU)	$=_2$
$m(\langle =_2, 2\rangle) = in$	(BF)	"."
$m/\langle \mathsf{a}, 1\rangle\langle ., 1\rangle = m/\langle =_2, 2\rangle\langle ., 1\rangle$	(BV)	A
$m/\langle \mathsf{a}, 1\rangle\langle ., 2\rangle = m/\langle \mathsf{a}, 1\rangle$	(BV)	X
$m/\langle \mathsf{a}, 2\rangle = m/\langle \mathsf{a}, 2\rangle$	(BV)	Y
$m/\langle \mathsf{a}, 3\rangle = m/\langle =_2, 1\rangle$	(BV)	Z0
$m/\langle =_2, 2\rangle\langle ., 2\rangle = \overline{m/\langle \mathsf{a}, 3\rangle}$	(BV)	Z

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

これらは，論理式の集合として扱うことも可能だが，モードグラフとして表現すると，表現と操作の双方に好都合である．モードグラフとは，

1. グラフの路 (path) が P_{Atom} の要素に対応し，
2. パス p に対応する節点は $m(p)$ の値を表現し，
3. 辺は，関数 / 述語記号と引数位置との対でラベルづけされるとともに，その辺を通るパスのモード値の解釈を反転する「反転記号」(図では ● 印) を持つこ

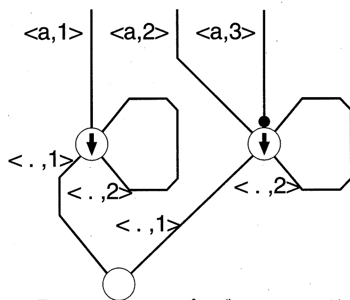

図2 append プログラムのモードグラフ

とができ，

4. $m/p_1 = m/p_2$ または $m/p_1 = \overline{m/p_2}$ の形の制約は節点の共有で表現した

ような素性グラフ (サイクルを許す素性構造 (feature structure)) [1]のことである．図2 が append プログラムから得られるモードグラフである．図の下向き矢印はモード値 in を表している．

モード解析は基本的に，個々のモード制約を表現する簡単なモードグラフを素性グラフの単一化によって次々と併合してゆく作業である．したがって，モード解析の決定可能性は，素性グラフの単一化アルゴリズムの決定可能性から保証される [14]．図1 の規則 (BV) は3個以上のサブモード間の制約を課することもあるが，ほとんどすべての場合，それらはモードグラフで表現可能な2個以下のサブモード間の制約に簡約化できる．

また解析の手間は，プログラムの記号数を n，各述語の各引数に対応する節点を根とするモードグラフの大きさの最大を d (d は，プログラムで使用する通信プロトコルの複雑さを反映する) としたとき，$O(nd \cdot \alpha(n))$ (α は Ackermann 関数の逆関数) であることがわかっている [1]．ただしこれは，well-moded なプログラムのモードグラフを作成する (もしくは well-moded でないことを検出する) 手間であり，プログラムが well-moded でない原因を解析する手間は含まない．

モード解析システムを含むかなり大きなプログラムに対してモード解析を試みたところ，モードグラフはプログラムが大きくなるにつれて大きくなるものの，その主因は使用する述語数が増えることにあり，d の値は，かなり複雑な通信プロトコルを持つプログラムでも，それほど大きくならない (高々数十程度) ことが観察された．つまり，大きなプログラムのモードグラフは，幅広く，浅いグラフとなることが一般に予想される [15]．したがって解析の手間は，プログラムの (記号数による) 大きさに対して，ほぼ線形オーダーである．

並行論理プログラミングでは，もっとも単純な型の概念は，関数記号 (定数を含む) の集合 Fun を，重なりを持たないいくつかの集合 $F_1, \ldots, F_n$ に分類することによって導入される．この型は，パスの集合 P_{Atom} から $\{F_1, \ldots, F_n\}$ への関数として定式化できる．矛盾なく行なわれるこのような型付けを well-typing と呼び，またこのように型付け可能なプログラムを well-typed なプログラムという．モード解析と同様，データ型の解析は，素性グラフの単一化問題として計算できる．Moded Flat GHC の節が課する型制約規則を図3 に示す．ただし，この $F_1, \ldots, F_n$ の選択の仕方としては任意のものが考えられ，その意味で，並行論理プログラミングではモード体系が型体系よりも基本的であるといえる．

3 静的解析による誤り箇所の同定

ここでは，Kima が利用しているプログラム中の誤り箇所の特定技法について概説する．詳細については [5] [6] [7]を参照されたい．

並行論理プログラムに誤りが存在する場合，そのプログラムは通信プロトコルの一貫性を失い，モード制約集合が充足不可能 (ill-moded) となることが多い (ただし，必ず充足不可能になるわけではない)．具体的には，プログラム中のあるデータ構造部 (パス) に関する正しい仕様を表したモード制約と誤ったそれとの衝突が，モード矛盾という形で現れる．これは，プログラム中の誤った記号出現が本来の仕様と異なった制約を課すために起こる．

このとき，モード制約集合の矛盾する極小部分集合を求めることで，矛盾の原因となっているプログラム中の節と記号を絞り込むことができる [5] †1．なぜなら，この極小集合の中には，誤って課された制約が少なくとも1つは存在しているはずであり，また，モード制約は節中の記号出現に対して課されるため (図1 のモードづけ規則を参照)，矛盾する極小部分集合の中の制約が課される原因となった節および記号出現を，誤りの原因の候補

†1 ここで示すアルゴリズムは，C が無矛盾だったときのことを考慮して，文献のものから改訂されている．

(HBF$_\tau$) h または B 中のパス p に関数記号が現れるならば $\tau(p) = F_i$
(HBV$_\tau$) h または B 中のパス p と p' に同じ変数が現れるならば $\tau/p = \tau/p'$
(GV$_\tau$) 同じ変数が h 中の p と G 中の p' に出現するならばどのパス q についても $\big(m(p'q) = in \Rightarrow \tau(pq) = \tau(p'q)\big)$
(BU$_\tau$) $=_k$ をボディの単一化ゴールとすると $\tau/\langle =_k, 1\rangle = \tau/\langle =_k, 2\rangle$

図3 節 h :- G | B が課する型制約

と見なすことができるからである．

型に関しても，モード制約と同様，プログラム中に誤りがある場合は，型制約の集合が充足不可能 (ill-typed) となることが多く，型に関する制約集合の矛盾する極小部分集合を求めることで，バグ箇所の候補を同定することができる．ただし，モードと型はそれぞれプログラムの性質に関する異なる抽象領域を表現しており，これらを併用することで誤りの検出力および修正案の品質を向上させることができる (4.2 節).

矛盾する極小部分集合は，図4に示す簡単なアルゴリズムを使って効率良く計算することができる．

ここで，$C = \{c_1, \ldots, c_n\}$ は制約のマルチ集合を表している．このアルゴリズムを適用することによって，C の中から求める極小部分集合 S が得られる．C が無矛盾であった場合は，$S = \{\}$ となる．また $false$ は，それ自身で矛盾する制約式であり，番兵として使われている．

このアルゴリズムによって得られる S の極小性の証明などについては，[5] を参照されたい．また，このアルゴリズムを適用した後に，求まった極小集合を除いた残りの制約集合にもう一度このアルゴリズムを適用することで，独立する複数の誤り (矛盾する極小部分集合) を一度に検出することが可能である．1つの誤りに対して，モードや型に関する極小部分集合は複数求まる可能性があるため，Kima はそれらをグループ化し，それぞれのグループに対して独立に修正案を探索する．これについては 4.3 節で詳しく説明する．

また，このアルゴリズムでは，集合演算による制約充足と，その結果である充足可能性のチェックは，第2節の素性グラフの単一化によって行なわれる．アルゴリズムは汎用のものであるが，その効率は制約充足系の効率

$c_{n+1} \leftarrow false$;
$S \leftarrow \{\}$;
while S が充足可能である **do**
  $D \leftarrow S$; $i \leftarrow 0$;
  while D が充足可能である **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** $S \leftarrow \{\}$

図4 矛盾する極小部分集合を求めるアルゴリズム

に大きく依存している．

矛盾する極小部分集合の平均サイズは比較的小さく，10より大きなものが求まることはほとんどない．つまり，極小部分集合の大きさは全体の制約の数とは関係がない．これは，2節で述べたように，モードグラフが広く浅い構造をしていることに起因している．極小部分集合の大きさは，述語呼び出しの依存関係の複雑さとそれに対応するモードグラフの深さに関係しているが，プログラムサイズの増大に対して，モードグラフの深さは極めてゆっくりとしか増加しない．またモードや型制約の冗長性から，制約の矛盾は，誤りを含む述語および依存関係の深い少数個の述語が課す制約集合の内部で発生することが多い．したがって，プログラムサイズが本技法に大きな影響を与えることはないといえる．もちろん人工的には，非常に大きな極小集合が求まるプログラムを作ることも可能である．

矛盾する極小部分集合の大きさが小さく抑えられてい

るため，バグの存在する可能性のある場所を，プログラムテキスト中の一定の範囲内に絞り込むことができる．

4 プログラムの自動修正

モード/型制約の矛盾する極小部分集合によって絞り込まれた，誤りの原因として怪しいとされる制約は，その原因となっている記号出現を書き換えることで，他の制約との衝突を解消することができる．具体的には，怪しいとされる制約を課した記号出現は，

- その記号出現を他の記号に書き換える，
  または
- その記号が変数記号である場合，同一節内の他の記号出現を書き換えることによって，その怪しいとされる変数記号の出現数を増やす (図1のモードづけ規則 (BV) を参照)

ことによって，他の制約に変化する可能性がある．ある記号を書き間違えた場合，それによって消滅した記号と新たに出現した記号が存在するわけだが，極小部分集合はこのどちらか (あるいは両方) を怪しい記号として特定している．つまり提案する枠組では，well-moded かつ well-typed なプログラムを「正しい」プログラムと考え，モードや型制約に関する矛盾をプログラムの軽微な書換えによって解消することで，プログラムの自動修正を実現する．

現在，Kima が修正の対象としているのは，抽象構文木 (abstract syntax tree) における終端記号の中でも，特に「変数の少数個の書き誤り」である．これは非常に限定的に聞こえるが，論理型言語ではプログラム中に変数を多用するために，単純な誤りの多くもそこから発生し，なおかつ，このような誤りを大きなプログラムの中から人間が自分で発見して直すのはそれほど容易ではない．

このとき，プログラムの自動修正は探索アルゴリズムによって実現され，

- 初期状態は誤りを含むプログラム，
- 目標状態は well-moded/typed なプログラム，
- 操作は変数の書換え

に対応づけられる．

たとえば，変数の1箇所の誤りを修正したい場合，仮に，矛盾する極小部分集合によってその場所が特定されていないとすると，プログラム中の記号のすべての書換えに対してモード/型解析を行ない，well-moded/typed の成否を検査しなければならない．しかし，誤り箇所を一定の狭い範囲内に絞り込んでいるため，その範囲内の書換えを考慮するだけで済み，探索空間の縮小と計算時間の節約を図ることができる．

モード/型制約は定数記号からも課されるため，本枠組は変数記号と定数記号との誤りに対しても適用可能である．型制約によって，定数記号どうしの誤りも修正できる場合がある．ただし，定数記号への書換えを考慮する場合は，データ型ごとに適当な値をあらかじめ決めておく必要があるが，それが常に正しいという保証はない．定数記号以外の関数記号への修正は可能性が大きくなりすぎるため，箇所の特定はできても，現在の Kima の枠組では修正は困難である．また，述語記号の誤りについては，本枠組を直接適用することはできない．

4.1 自動修正アルゴリズム

誤りが軽微であるという仮定から，修正案の探索は書換え個数に関する深さ漸増探索によって行なうのが合理的である．ただし現在の Kima は，ユーザの時間的な制約を考慮して，指定された深さまでの修正案を求めることにしている．また，Kima は修正案がどの深さの探索で求まったものであるかを区別しないため，必ずしも深さ漸増探索である必要はなく，深さ優先探索などを用いてもよい．ただし，許容時間内における最良の解を求めたい場合は，やはり深さ漸増探索が有効である．

Kima が採用している自動修正アルゴリズムを図5に示す．まずモードや型に関する (1つの) 矛盾する極小部分集合から誤りの原因を特定し，それを元にユーザによって指定された深さ (書換え個数) MAX までの変数の書換え案を網羅的に生成する．このとき，変数 *depth* が現在の深さに対応している．そして書き換えたプログラムのモードおよび型を検査することで修正案を求める．

書き換えたプログラムのモード/型を検査するには，プログラムサイズに比例する手間が必要となるため (2節)，修正案の探索効率はプログラムサイズに比例して悪化する．ただし，3節で述べたように，制約の矛盾は，誤りを含む述語および依存関係の深い少数個の述語が課す制約集合の内部で発生することが多い．このことを利

```
モード / 型制約の矛盾する極小部分集合を計算;
そこから疑わしい節および (変数) 記号を抽出;
depth ← 1;
while MAX ≥ depth do
 while depth 個の記号の書換え方がまだある
  do
  その depth 個の記号を書き換える;
  if 書き換えたプログラムが well-moded
    かつ well-typed になった
   then その書換えを修正案として出力
  fi
 end while;
 depth ← depth + 1
end while
```

図5 自動修正の基本アルゴリズム

用して，モード / 型検査を行なう際，まず最初に矛盾する極小部分集合に関与した述語 (や依存関係の深い一部の述語) から課される制約集合についてだけモード / 型検査を行ない，この検査に通るものに関してのみ制約集合全体との整合性を検査するようにすれば，効率をかなり改善できると考えられる．

修正案は一般に複数求まるため，修正案が正しいかどうかの確認は，最終的にはユーザに委ねられる．ただしKima による解析は，言語処理系とはまったく独立に行なわれ，また，ユーザのプログラムを直接書き換えるわけではないため，解析によって余計な混乱が生じることはない．

4.2 モード制約と型制約の併用

Kima は，モードと型に関してそれぞれ矛盾する極小部分集合を求める．これは，モードと型がそれぞれ通信プロトコルとデータ型というプログラムの持つ異なる性質を表現しており，検出できる誤りの種類が異なるためである．モード情報と型情報を併用することで検出率を向上できるだけでなく，誤りの可能性のある場所をより狭い範囲に絞り込むことができる．また，モードと型の併用は，求まる修正案の数を縮小し，修正案の品質を向上させるのにも有効である．

表1 Kima における型の分類

型	関数記号
F_1	整数
F_2	浮動小数点数
F_3	文字列
F_4	ベクタ
F_5	リスト
F_6	ストラクチャ

Kima が採用している強型体系では，関数記号を重なりのない6つの単純な型，すなわち，整数，浮動小数点数，文字列，ベクタ，リスト，ストラクチャに分類し，これらのうちの2つ以上の型が同一のパスを共有することを許さない (2 節)．これらをまとめたものを表1 に示す．ここで，ストラクチャ型は，KL1 における記号アトムおよびファンクタを表している．ただし，特殊な記号アトムである空リスト “[]” は，ストラクチャ型ではなく，リスト型に分類する．

これは，現実の多くのプログラムでは，同一のパスに表1 における複数のデータ型が同時には現れないことに基づいたヒューリスティックな分類であり，このような単純な分類であっても検出率の向上や修正案の品質の向上に有効であることが実験により確かめられている (表2)．

4.3 誤り箇所のグループ化

モードや型に関する矛盾する極小部分集合は，独立したものが複数求まる可能性があるが，それらが同一の節や (変数) 記号を誤りの原因として指摘している場合がある．たとえば，ある単一の誤りがモードと型両方に関する矛盾を起こしている場合は，モードと型両方の矛盾する極小部分集合が，制約を課した原因である節および記号を指摘することになる．このとき，指摘する節を共有する極小部分集合同士を，誤りに対する同一の「グループ」であるとみなし，修正案探索に関する単位にする (図6)．

グループ化は以下の2つの原則に従って，これ以上グループ化が進まなくなるまで行なう．

- 独立した異なる極小部分集合 X と Y が，同一の節

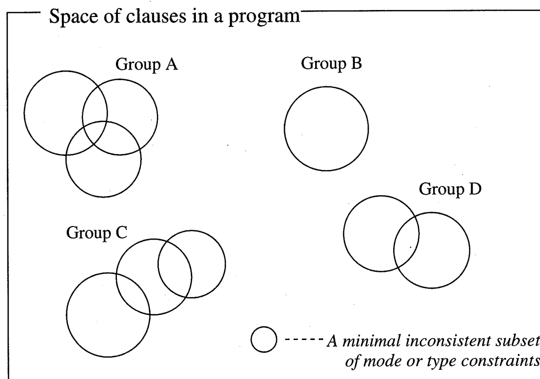

図 6　矛盾する極小部分集合のグループ化

を誤りの原因の候補として指摘している場合，X と Y を同一のグループにグループ化する．

- 3 つの独立する極小部分集合 X, Y, Z に対して，X と Y が同一グループであり，Y と Z が同一グループであるならば，X, Y, Z を同一グループにグループ化する．

そして，深さ *depth* の修正案の探索は，このグループごとに，所属する (複数の) 極小部分集合が指摘している節中の記号を *depth* 箇所書き換えることによって行なう．

このとき，ある *depth* 個の書換えがグループ内のすべての極小部分集合の矛盾を解消させる可能性があるかどうかを先にチェック (**クイックチェック**) することで，書き換えた後のプログラムに対するモード / 型解析の回数を減らし，計算時間を短縮することができる．

つまりクイックチェックとは，ある *depth* 個の書換えが，グループ内のすべての極小部分集合に関して，極小部分集合が誤りの候補として指摘している変数のうち少なくとも 1 つを

- 他の変数記号に書き換えているか，
  または
- 同一節内の他の変数記号を書き換えることで，その指摘された変数の出現数を増やす

ことになっているかどうかを調べることである．

グループ化とクイックチェックの手続きを追加した自動修正アルゴリズムを図 7 に示す．

モード / 型制約の冗長性が高く，矛盾する極小部分集合の中から誤りの原因である制約式をさらに絞り込める

(独立複数の) モード / 型制約の矛盾する極小部分集合を計算;
それらをグループ化する;
for each グループ **do**
　グループ内の極小部分集合から疑わしい節および記号を抽出;
　depth ← 1;
　while MAX ≥ *depth* **do**
　　while *depth* 個の記号の書換え方がまだある **do**
　　　if その書換え方がクイックチェックに通る
　　　　then
　　　　その *depth* 個の記号を書き換える;
　　　　if 書き換えたプログラムが well-moded かつ well-typed になった
　　　　　then その書換えを修正案として出力
　　　　fi
　　　fi
　　end while;
　　depth ← *depth* + 1
　end while
end for

図 7　グループ化を用いた自動修正アルゴリズム

状況 [7]では，ほとんどの場合，同一グループに属する複数の極小集合が求まるため，グループ化およびクイックチェックは文献 [7]における絞り込みと同等の効果がある (ただし厳密に同じではない)．

5　モード / 型以外の情報の利用

Kima は，モードや型に関する情報の他に，簡単な構文上の規則やヒューリスティクスを利用することで，検出力の強化や修正案の品質の向上，さらに，探索の効率化を実現している．

5.1　修正案への優先度づけ

第 4 節で述べたように，Kima はモードおよび型情報を用いて修正案を探索するが，このとき修正案は一般

に複数求まる．そこでKimaは以下のような**ヒューリスティクス**を使い，求めた修正案に優先度をつけることで，修正案の品質を向上させている．

ヒューリスティクス 1 節中において，ある変数が

1. singleton (1回のみ) 出現
2. ヘッドで2回以上出現
3. ヘッドまたはボディに3回以上出現
4. ある述語呼び出しの引数に2回以上出現

しているようなものは，もっともらしくない．

ヒューリスティクス 2 リスト自身とその要素の型が同じ，すなわち，プログラム中のある節において，パス p と要素のパス $p\langle.,1\rangle$ (非空リストの第1引数目) に同一の変数が出現している場合はもっともらしくない．

ここで「もっともらしい (らしくない)」は，「プログラム中の変数を少数個だけ書き換えた (モード / 型の正しい) プログラムの集合を考えたとき，それを極力守っているものの方が，より意図したプログラムに近い (遠い) であろう」という意味である．

ヒューリスティクス 1.1, 1.2, 1.3 のような変数出現は，IN, OUT といった in, out よりも強いモード制約を課すことが多いため (2節)，ヒューリスティクス 1.1, 1.2, 1.3 は「より弱いモード制約が課せられる方がもっともらしい」と言い換えることもできる．モードや型に関する制約が強化または追加されると，プログラムが実行時に満たさなければならない条件が厳しくなり，それだけプログラムが実行に失敗する可能性が高くなる．したがって，プログラムに課せられる制約は弱ければ弱いほどよいといえる．

現実のプログラムでは，論理変数は1対1通信に用いられる場合が大変多い [15] という事実からもヒューリスティクス 1.1, 1.3 は理由づけできる．1対1通信に用いられる変数は，節中のヘッドとボディにちょうど1回ずつ，またはボディにちょうど2回出現する．ヒューリスティクス 1.4 の状況は，他の述語からデータを重複して受信しているか，あるいは自分自身と通信していることになり，もっともらしくないといえる．

ヒューリスティクス 2 で述べているプログラムには，α を型変数とし，α 型のデータを要素とするリストの型を $list(\alpha)$ と書くことにしたとき，$\alpha = list(\alpha)$ というデータ構造に関する制約が追加されることになる．これは α に関する強い型制約を課すという点でもっともらしくない．

Kimaは，得られた複数の修正案に対し，これらの「もっともらしくない」部分に一定のペナルティを与えることで修正案に優先度をつける．

5.2 検出率の向上

Kimaは，プログラムに関する仕様や宣言によらず，モードと型情報だけからプログラム中の誤りを検出するため，誤りが必ず検出されるわけではない．実験では，1箇所の誤りに対する検出率は 70% 程度にとどまっている (表2)．そこでKimaは**検出規則**を設けて検出力の強化を図っている．検出規則を以下に示す．

検出規則 1

1. ガードで検査している変数がヘッドになくてはならない
2. ユニフィケーションの両側に同一の変数が出現してはならない (部分的な occur-check)

検出規則 2 singleton 出現している変数の名前は，下線 “_” で始まっていなければならない

Kimaは，**検出レベル** (6節) をオプションとして指定することで，誤りの検出の際，検出規則をユーザが選択的に利用できるようにしている．

プログラムの正規化 [14] の視点から考えると，検出規則 1.1 への抵触はガードゴールの消滅を意味し，検出規則 1.2 への抵触は正規化の失敗を意味する．

変数の singleton 出現は，変数が持っているデータが破棄されることを意味しており，検出規則2はそのような変数をプログラマに宣言させることと等しい．

また，プログラムの読みやすさの点からも利点が大きいといえる．論理型言語 Prolog の主な処理系も同様の警告を発する．

論理変数は節中にちょうど2回出現する場合が大変多いため，ある変数を書き誤ったときには，その変数の残りの出現が singleton 出現になる可能性が非常に高い．そのため，検出規則2を守ってプログラムを作成することで，モード / 型情報からは発見されない多くの誤りを検出できるようになる．

```
モード制約の矛盾する極小部分集合を計算;
そこから疑わしい節および記号を抽出;
検出規則に抵触する節と記号の検出;
depth ← 1;
while MAX ≥ depth do
  while 指定された優先度以上の depth 個の
        変数の書換え方がまだある do
    その depth 個の記号を書き換える;
    if 書き換えたプログラムが検出規則に抵触
      しない then
      if 書き換えたプログラムが well-moded
         かつ well-typed になった
       then その書換えを修正案として出力
      fi
    fi
  end while;
  depth ← depth + 1
end while
```

図8 自動修正アルゴリズム (改良版)

このように，検出規則は場当たり的なものではなく，合理的な理由に基づいて規定されている．

検出規則によって検出される誤りの原因はある節の変数記号であり，モードや型制約の矛盾する極小部分集合とは独立して求まることになる．Kima は，検出規則によって検出された変数記号を要素数1の独立した極小集合とみなして，モード / 型制約の矛盾する極小部分集合と一緒にグループ化することで，誤りの候補を統一的に扱っている．

5.3 修正案探索の最適化

5.2 節の検出規則は節中の変数の出現の仕方を調べるだけで済み，一度検査して異常がないと判断された節については再度検査を行なう必要がない．矛盾する極小部分集合が指摘する，プログラム中のごく一部の節を書き換えたとき，節の大きさはプログラムの大きさに関係なくほぼ一定であるので，検出規則による再検査は一定の手間しか必要としない．

それに対し，モード / 型の検査の場合は，書換えを行なった節とプログラム全体の整合性がとれているかどうかを調べるために，プログラム全体のモード / 型を再計算する必要がある．この作業には，プログラムサイズに(ほぼ) 比例する手間がかかる (2 節， 4.1 節)．

したがって，モード / 型解析の前に検出規則を使った検査を行なうことでモード / 型解析の回数を減らし，探索の効率を向上することができる．

また，ヒューリスティクス1による優先度づけに関しても同様で，優先度をつけるには節中の変数出現を調べるだけで済む．したがって優先度の高い修正案だけを求めたい場合は，モード / 型解析による検査の前に優先度づけを行ない，優先度の高いものだけをモード / 型解析するようにすれば，同じく探索の効率を向上できる．

プログラムサイズが大きいほど，モード / 型解析の回数が減ったことによる効率の向上も大きくなるはずである．ただし，ヒューリスティクス2による優先度づけについては型解析が必要となるため，従来どおり修正案が求まった後に行なうことにする．

これら2種類の最適化を使った自動修正アルゴリズムを図8に示す．ただし簡単のため，ここではグループ化に関する手続きは省略している．

同一節中に2箇所の変数の誤りを含んだクイックソートプログラムの例では，この最適化により，もっとも優先度の高い修正案を求めるのにかかった時間が Sun Ultra 30 (248 MHz) + 128 MB メモリ上で 25.9 秒から 10.2 秒に短縮された．

6 実験結果

いくつかの実験を通して，本技法の有効性について論じる．調査したのは，プログラム中に数箇所の誤りがあった場合の，検出率や修正案の数，さらに，元となる正しいプログラムの周辺に存在する「もっともらしい」プログラムの数である．

6.1 実験1

ここでは，プログラム中の変数の1箇所の書き間違いを網羅的につくり，これまでに述べてきた自動修正技術を適用して，検出成功数と，求まる修正案の数の比較を

表2 変数の1箇所の書き間違いに対する検出成功数と求まる修正案の数

プログラム	利用情報	検出レベル	優先度づけ	実験総数	検出成功	求まる修正案の数 1	2	3	4	5	6	7	≥8
append	モードのみ	0	なし	58	36	1	3	8	3	6	5	3	7
	型のみ	0	なし	–	0	0	0	0	0	0	0	0	0
	モード＆型	0	なし	–	36	1	3	8	3	6	5	3	7
	〃	0	あり	–	36	27	9	0	0	0	0	0	0
	〃	1	なし	–	40	2	12	4	3	5	6	4	4
	〃	1	あり	–	40	29	11	0	0	0	0	0	0
	〃	2	なし	–	58	21	12	3	3	9	7	3	0
	〃	2	あり	–	58	39	19	0	0	0	0	0	0
fibonacci	モードのみ	0	なし	118	57	10	7	9	6	4	1	9	11
	型のみ	0	なし	–	47	0	0	4	20	0	18	0	5
	モード＆型	0	なし	–	72	18	13	2	15	9	0	6	9
	〃	0	あり	–	72	54	11	1	6	0	0	0	0
	〃	1	なし	–	88	34	16	11	15	7	2	3	0
	〃	1	あり	–	88	68	12	7	0	1	0	0	0
	〃	2	なし	–	99	58	6	9	11	5	5	1	4
	〃	2	あり	–	99	71	18	8	0	2	0	0	0
quicksort	モードのみ	0	なし	300	177	34	70	1	12	19	0	12	29
	型のみ	0	なし	–	106	0	2	12	40	0	32	0	20
	モード＆型	0	なし	–	221	49	76	8	59	0	9	18	2
	〃	0	あり	–	221	164	41	16	0	0	0	0	0
	〃	1	なし	–	236	76	98	20	19	0	7	16	0
	〃	1	あり	–	236	175	61	0	0	0	0	0	0
	〃	2	なし	–	286	142	101	21	16	3	0	2	1
	〃	2	あり	–	286	199	84	2	1	0	0	0	0

行なった．その結果を表2に示す†2．

実験に用いたプログラムは，以下の3つである．

append:

```
1: append([],    Y,Z ) :- true | Y=_1Z.
2: append([A|X],Y,Z0) :- true |
3:     Z0=_2[A|Z], append(X,Y,Z).
```

fibonacci:

```
1: fib(Max,_, N2,Ns0) :- N2 >Max | Ns0=_1[].
2: fib(Max,N1,N2,Ns0) :- N2=<Max |
3:     Ns0=_2[N2|Ns1], N3:=N1+N2,
4:     fib(Max,N2,N3,Ns1).
```

quicksort:

```
1: quicksort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
2: qsort([],     Ys0,Ys ) :- true | Ys=_1Ys0.
3: qsort([X|Xs],Ys0,Ys2) :- true |
4:     part(X,Xs,S,L), qsort(S,Ys0,[X|Ys1]),
5:     qsort(L,Ys1,Ys2).
6: part(_,[],  S, L ) :- true | S=_2[], L=_3[].
```

†2 文献 [3] における同種の実験では，ガードの書き誤りや検出レベル1に関連する誤りについてカウントしていないほか，型誤りによって新たに発見された誤りについては修正案を求めていないなどの理由から，実験総数や修正案数等の値が異なっている．

```
 7: part(A,[X|Xs],S0,L ):- A>=X |
 8:      S0=4[X|S],part(A,Xs,S,L).
 9: part(A,[X|Xs],S, L0):- A< X |
10:      L0=5[X|L],part(A,Xs,S,L).
```

いずれも簡単なプログラムであるが，これは網羅的な実験によって，提案する技法の基本的な能力を明らかにするためである．実用規模のプログラムを考えた場合でも，プログラムサイズが本技法の有効性に深刻な影響を与えることはない (4.1 節).

実験で用いているのは，述語の定義部分だけであり，述語の呼び出し部分については利用していない．当然のことながら，利用した方が制約の冗長性が高くなるため，検出率や修正案の品質の向上には有利である．また，変数名を "_" で始まる変数と書き誤るような間違いについては考慮していない．

検出レベル 0 では検出にモード / 型情報だけを使い，レベル 1 ではそれに加えて検出規則 1，レベル 2 では検出規則 1 と 2 の両方を使っている．レベル 2 の検出規則を用いることで，モードと型情報のみを利用する場合と比べ，検出率の平均が 69.1% (329/476) から 93.1% (443/476) へと大幅に向上することがわかる．

「優先度づけあり」の行は，優先度のもっとも高いものだけを修正案として求めた場合の修正案数を表している．ほとんどの場合，意図通りの正しい修正がもっとも優先度の高い修正案の中に含まれており，求まる修正案の数を正しい修正を含む 1 つかまたはごく少ない数に絞り込むことに成功している．その例外は `fibonacci` プログラムのごく一部の例である．`fibonacci` プログラムは，節中に 4 回出現する変数が存在する少し稀な例であるため，そのうちの 2 つの出現がその節に現れないまったく新しい変数で置き換わるという修正案がもっとも優先度の高い修正案として求まり，意図通りの修正案の優先度が低くなってしまう場合があった．

6.2 実験 2

次に，同一節中の 2, 3 箇所の誤りに対する検出率を表 3 に示す．この種の誤りは必ず同一グループ (4.3 節) における誤りとなり，修正案は深さ 2, 3 の探索によって求められる (後述するプログラムの等価性のために，深さ 1 の探索で見つかる場合もある)．逆に，節の異なる複数箇所の誤りは，その節がたとえ同じ述語の定義節であっても，矛盾する極小部分集合の求まり方によっては異なるグループに分類され，それぞれのグループにおける 1 箇所の誤りとみなされる場合がある．

つまり，ここで調べているのは基本的に深さ 2, 3 の探索が必要となるような誤りに対する検出率であり，誤りが異なる節にまたがっているような場合については調査していない．また，ここでも実験 1 と同様，変数名を "_" で始まる変数と書き誤るような間違いについては考慮していない．

表 3 の実験総数や検出成功数では，「書換え方は異なるが，結果的に等価なプログラムになる」ようなものの除去は行なっていない．同一節中の N 箇所の誤りを網羅的に生成して，検出に成功するものを単純にカウントしているだけであり，実験総数と検出成功数のどちらにも等価なプログラムが複数存在している．ここでいう「等価なプログラム」とは，

1. 変数名が異なるだけのプログラム (α 同値)
2. 単一化述語などのように交換則 (`A=B` $\Leftrightarrow$ `B=A`) が成り立つ 2 引数の組込み述語に対して，引数の順番を変えただけのプログラム

のことを指す．

節中に複数の誤りが存在する場合，そのうちの 1 つでも誤りを起こす原因があれば誤りの検出に成功するため，誤りが 1 箇所である場合 (表 2) よりも結果として検出率が高くなっている．特に検出レベル 2 を利用した場合は，いずれの例でも検出率が 95% を上回ることがわかった．

6.3 実験 3

最後に，同一の節中の変数を N 箇所書き換えたとき，モード / 型や検出規則を通るだけでなく，元の正しいプログラムと同じかそれ以上の優先度を持つプログラムが何通りあるかを調べた．その結果を表 4 に示す．ここではこれまでの実験 1, 2 とは異なり，"_" で始まる変数への書換えを考慮している．また，種類数の列では，書換え方は異なるが等価なプログラムになるようなものをまとめて 1 つとみなしている．

実験を行なった範囲では，N が増加しても同等以上の優先度を持つプログラムの種類は爆発的には増えて

表3 節中の変数を N 箇所書き換えた場合の誤り検出率

プログラム	N	検出レベル	実験総数	検出成功	検出率(%)
append	2	0	1200	937	78.1
	2	1	–	1004	83.7
	2	2	–	1141	95.1
	3	0	16980	14597	86.0
	3	1	–	15411	90.8
	3	2	–	16674	98.2
fibonacci	2	0	4668	3982	85.3
	2	1	–	4330	92.8
	2	2	–	4489	96.2
	3	0	133045	125300	94.2
	3	1	–	130325	97.9
	3	2	–	131810	99.1
quicksort	2	0	12102	11263	93.1
	2	1	–	11460	94.7
	2	2	–	12005	99.2
	3	0	337455	330769	98.0
	3	1	–	332416	98.5
	3	2	–	336943	99.8

いない．N がさらに増えても同様であろうと予想される．これは，変数の任意の配置の仕方に比べて，well-moded/typed かつ変数がちょうど2回出現するような変数の置き方が極めて少ないためである．後者が一定数しか存在しないために，N が増えるほど総数に対する種類数の割合は大きく減少することになる．

ここで，修正時に求まる修正案数について考える．仮に，プログラムが同一節中に2箇所の変数の誤りを含んでおり，その節に対して深さ2の(2箇所の書換えによる)修正案の探索が実行されたとする．この場合，元の正しいプログラムから考えると，最大で4箇所の変数が書き換えられる可能性があり，N=4 であるような書換え案がもっとも多くなる．しかし，最初の2箇所の書換えは，誤って書き換えたものとしてすでに決定しているため，実際に書換え案として生成されるのは，その中のごく一部である．実際に生成される書換え案の数は N=2 の場合の総数と(ほぼ)同じである．完全に同じになるとは限らないのは，書き誤りによって節中の変数の種類数が変化する可能性があるためである．

表4 節中の変数を N 箇所書き換えたプログラムの中で同等以上の優先度を持つものの種類数

プログラム	N	実験総数	種類数
append	1	58	0
	2	1200	7
	3	16980	14
	4	167842	29
fibonacci	1	118	11
	2	4668	66
	3	133045	309
quicksort	1	300	9
	2	12102	33
	3	337455	76

append の例でいえば，4箇所を書き換える書換え方は 167,842 通りあり，その中で意図通りのプログラムと同等以上の優先度を持ったプログラムの数は29個であるが，その中で実際に探索されるのは N=2 の場合の 1,200 通りほどである．

「もっともらしい」プログラムのうち，(発散や失敗をせずに)停止するプログラムの割合は，元のプログラムの特徴やプログラムの期待する入力の種類によって異なる．クイックソートプログラムの場合，停止するプログラムの割合は約半数であった．また，その中で，人間が見て意味があると思える，つまり，すべてのオペレーションが計算結果に貢献しているようなプログラムは，ごく一部であった．

7 アルゴリズムの適用例

7.1 例1 — Append

例として，変数を1箇所書き間違えている append プログラムを考える．

1: R_1 : append([],　Y,Z):- true | Y=$_1$Z.
2: R_2 : append([A|Y],Y,Z0):- true |
3: 　　　Z0=$_2$[A|Z], append(X,Y,Z).

(R_2 節の頭部は append([A|X],Y,Z0) が正しい)

このプログラムは ill-moded となり，3節のアルゴリズムを適用することによって，以下のモード制約に関

する極小部分集合が得られる．

モード制約	規則	原因
$m/\langle \mathrm{append}, 1\rangle\langle .\,, 2\rangle = IN$	(HV)	Y in R_2
$m/\langle \mathrm{append}, 1\rangle = OUT$	(BV)	X in R_2

これが得られたことにより，R_2 節の2つの変数 X と Y が誤りの原因として怪しいことがわかる．実際，X と書くべきところを Y と書き間違えているので，この指摘は正しい．多くの場合，書き間違える前の正しい変数か間違えた後の変数のどちらかしか指摘されないが，この例の場合はその両方が指摘されている．

また，型に関する矛盾は発生しない．これは，append プログラムに現れる関数記号が，すべてリスト型に属しているためである．しかしこのような場合でも，モードに関する制約は，誤りの原因となっている記号の検出に成功している．

また，検出規則2を用いた場合は，X が R_2 節において singleton 出現していることから，変数 X が誤りの原因として怪しいことがわかる．

まず1箇所の書換えによって修正案を探索することを考える．この場合，R_2 節における変数 X または Y を他の変数に書き換えるか，または他の変数を X や Y に書き換えることによって修正案を探索する．検出規則2を用いた場合は X に関する書換えだけを考慮すればよい．その結果，次の6つの修正案が解として求まる．

(1) Line 2: append([A|X],Y,Z0) :- true |
(2) Line 2: append([A|Y],X,Z0) :- true |
(3) Line 3: Z0 $=_2$ [A|Z], append(Y,Y,Z).
(4) Line 3: Z0 $=_2$ [A|Z], append(Z0,Y,Z).
(5) Line 3: Z0 $=_2$ [A|Z], append(A,Y,Z).
(6) Line 3: Z0 $=_2$ [A|Z], append(Z,Y,Z).

修正案 (3), (4), (5), (6) は，変数 Y の頭部における複数出現や，その他の変数の3回以上の出現のために，ヒューリスティクス1によって優先度が下げられる．実際，これらの修正案に基づくプログラムは，ほとんどの入力に対して実行に失敗する．また，修正案 (5) はヒューリスティクス2によってさらに優先度が下がる．

結果として残った修正案 (1), (2) のうち，(1) がユーザの意図通りの append プログラムである．しかし残った修正案 (2) は，2つのリストの要素を交互にマージするプログラムになっている．たとえば append([a,b,c],[d,e,f],Res) として呼び出したとき，(1) では [a,b,c,d,e,f] が結果として求まるが，(2) では [a,d,b,e,c,f] が求まる．

9節で今後の課題として述べる「正しい入力と出力の例」が利用できれば，このようなものを除去することができるが，これは仕様を与えない範囲内での近似的に正しいプログラムであるといえる．

7.2 例2 — Fibonacci sequence

次に，フィボナッチ数列を求めるプログラムの，同じく変数を1箇所書き間違えている例を考える．

```
1: R1 : fib(Max,_, N2,Ns0) :- N2 >Max |
2:          Ns0 =1 [].
3: R2 : fib(Max,N1,N2,Ns0) :- N2=<Max |
4:          N1 =2 [N2|Ns1], N3:=N1+N2,
5:          fib(Max,N2,N3,Ns1).
```

(4行目の単一化ゴールは Ns0 $=_2$ [N2|Ns1] が正しい)

このプログラムはモードと型両方に関して矛盾を発生するため，それぞれについて矛盾する極小部分集合が求まる．検出規則2を利用している場合は，R_2 節の変数 Ns0 のヘッドにおける singleton 出現も誤りとして検出されるが，簡単のためここでは省略する．

モードに関しては，矛盾する極小部分集合が独立して2つ求まる．

極小部分集合1 (モード):

モード制約	規則	原因
$m(\langle =_1, 2\rangle) = in$	(BF)	"[]" in R_1
$m/\langle =_1, 1\rangle = m/\langle \mathtt{fib}, 4\rangle$	(BV)	Ns0 in R_1
$m/\langle =_1, 2\rangle = \overline{m/\langle =_1, 1\rangle}$	(BU)	$=_1$ in R_1
$m(\langle \mathtt{fib}, 4\rangle) = IN$	(BV)	Ns0 in R_2

極小部分集合2 (モード):

モード制約	規則	原因
$m(\langle =_2, 2\rangle) = in$	(BF)	"." in R_2
$m/\langle =_2, 2\rangle = \overline{m/\langle =_2, 1\rangle}$	(BU)	$=_2$ in R_2
$m(\langle =_2, 1\rangle) = IN$	(BV)	N1 in R_2

型に関しては，次の極小部分集合が1つだけ求まる．

極小部分集合 3 (型):

型制約	規則	原因
$\tau/\langle \texttt{fib}, 2\rangle = \tau/\langle \texttt{:=}, 2\rangle\langle \texttt{+}, 1\rangle$	(BV_τ)	N1 in R_2
$\tau(\langle \texttt{=}_2, 2\rangle) =$ リスト型	(BF_τ)	"." in R_2
$\tau/\langle \texttt{fib}, 2\rangle = \tau/\langle \texttt{=}_2, 1\rangle$	(BV_τ)	N1 in R_2
$\tau/\langle \texttt{=}_2, 2\rangle = \tau/\langle \texttt{=}_2, 1\rangle$	(BU_τ)	$\texttt{=}_2$ in R_2
$\tau(\langle \texttt{:=}, 2\rangle\langle \texttt{+}, 1\rangle) =$ 整数型	builtin	:= in R_2

極小部分集合 1, 2, 3 はどれも，怪しい節として R_2 節を指摘しており，同じグループに分類される．指摘する節と変数記号についてまとめると次のようになる．

節	変数記号	指摘する極小部分集合
R_1	Ns0	1
R_2	Ns0	1
	N1	2, 3

まず 1 箇所の書換えによる修正案の探索を行なうことにする．このとき，R_1 節において，変数 Ns0 を他の変数に書き換える，あるいは Ns0 の出現を増やすといった書換えは，それによって解消されうる極小部分集合が 1 だけであり，矛盾する極小部分集合すべてについて要素である制約が変化する可能性がない．クイックチェックを行なうことで，そのような書換えに対してはモードや型の検査を行なわなくてよいことが即座にわかる．

そこで R_2 節における書換えを考えると，1 箇所の書換えによってすべての矛盾する極小部分集合を解消しうるような書換え方は，Ns0 を N1 に書き換えるか，あるいはその逆 (N1 を Ns0 に書き換える) だけである．それ以外の書換え方では，すべての極小部分集合の矛盾を解消することができない．R_2 における Ns0, N1 の記号出現は合計 4 個であるので，それぞれを置換する 4 通りのプログラムに関してだけモード / 型解析を行ない，制約全体が充足するか否かを調べればよいことになる．

その結果，次の修正案がただ 1 つだけ求まる．

(1) Line 4: `Ns0=`$_2$`[N2|Ns1],N3:=N1+N2,`

これはユーザの意図通りの正しい修正になっている．

8 関連研究

論理式によって表現された系の誤動作の原因を系統的に解析する技術は，人工知能分野でモデルベースの診断として知られており [11]，多重故障の解析，極小の説明の探索などの特徴に共通点が見られる．しかしその目的は，系の本来の仕様 (first principle) を与え，それと観測された挙動との差異を解析することにある．(部分的な) 仕様と実際の挙動との差異を使ってプログラムの誤りを検出する技法は，宣言的デバッグと呼ばれ，宣言や表明 (assertion)，ユーザからの対話的な情報などを用いてデバッグの支援を行なうものである [8] [9] [12]．型宣言もこの一種であるとみなすことができる．本研究の枠組は，プログラムの本来の仕様を陽に与えることなく誤りの箇所を特定できるのが大きな特徴であり，その点でこれらの研究とは異なっている．

関数型プログラムにおける型誤りの診断技法は [16] に見ることができる．これは制約充足問題を単一化問題として解く際に，単一化アルゴリズムを拡張することによって，単一化に失敗したときの原因を提示できるようにしたものであるが，本論文の技法は，制約充足の一般的枠組みのレベルで診断方式を検討することにより，単一化アルゴリズム自体に変更を施すことなく誤り診断を実現している．このため，他のシステムへの適用性の面で有利である．また誤りの箇所を，誤りを含む極小部分集合またはそれよりも狭い範囲内に絞りこむことに成功している．

また，型体系を利用して関数型プログラムの自動的な修正を試みているものとして [13] がある．これは，ソフトウェア開発工程において，ソフトウェア部品 (コンポーネント) の 1 箇所の変更を行なったときに，その変更が正しく行なわれることを前提に，変更によって矛盾が発生する部品を検出し，変更に適合する他の既存のコンポーネントの検索および交換を自動的に行なうものである．Kima は，どこに誤りが存在するかわからない状態から誤りの箇所を静的に特定し，探索によってその自動修正を行う．また，Kima が誤りの対象としているのは変数記号であり，コンポーネントよりも低レベルのコンストラクトである．

9 まとめと今後の課題

強モード / 型体系の下で，プログラムに関する明示的な宣言や仕様を要求することなく並行論理プログラムの自動修正を効率的に行なうための技術を開発し，ソフト

ウェア Kima 上でそれを実現した．

Kima を用いていくつかの定量的な実験を行なった結果，多くの場合，KL1 (Moded Flat GHC) プログラム中の変数の少数個の誤りに対して，正しい修正を含む1つあるいはごく少ない数の修正案を提示できることがわかった．これにはモード / 型情報のほかに，理論的または統計的に裏打ちされたヒューリスティクスの併用が大きく貢献している．本論文で示した検出規則やヒューリスティクスは検出力の強化や修正案の品質の向上だけでなく，修正案探索の効率化にも有効である．そして，プログラム中の変数を数箇所書き換えたプログラムの集合を考えたとき，その中で well-moded/typed かつ高い優先度を持つようなものは全体に対してごく一部であることが確認できた．

今後の課題として．プログラムに関する仕様や宣言を与えることで，より高度な修正を行なうことが考えられる．たとえば，プログラムの正しい入出力例はモードや型に関する仕様にもなっており，これを利用することで誤り検出や修正案探索の手間が大幅に縮小できると予想される．また，層化による多相モード / 型述語への対応 [5] については現在実装中である．

Kima はそれ自身 KL1 を使って記述されており，現在の大きさは約 4500 行である．100 行程度のプログラムであれば，矛盾する極小部分集合の計算や深さ1の探索は数秒で完了することができる．7.1 節の例では，Sun Ultra 30 (248 MHz) + 128 MB メモリを用いて 0.05 秒以下で修正案が求められる．Kima の持つ修正技術の実用性はまだ実験段階であるものの，誤りの検出およびその箇所の特定能力は Kima 自身の開発に大変有用であった．

謝辞

本論文の改善に関して貴重なご意見をくださいました査読者の方々に深く感謝致します．

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] 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.

[3] 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.

[4] Chikayama, T., Fujise, T. and Sekita, D. : A Portable and Efficient Implementation of KL1, In *Proc. Sixth Int. Symp. on Programming Language Implementation and Logic Programming (PLILP '94)*, LNCS 844, Springer, 1994, pp. 25–39.

[5] 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.

[6] 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, 並列処理シンポジウム *(JSPP '96)* 論文集, 情報処理学会, 1996, pp. 219–226.

[7] 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, 1996, pp. 37–40.

[8] Shapiro, E. : *Algorithmic Program Debugging*, ACM Distinguished Dissertation Series, The MIT Press, 1982.

[9] Fromherz, M. : Towards Declarative Debugging of Concurrent Constraint Programs, In *Proc. First Int. Workshop on Automated and Algorithmic Debugging (AADEBUG '93)*, LNCS 749, Springer, 1993, pp. 88–100.

[10] Milner, R. : A Theory of Type Polymorphism in Programming, *J. of Computer and System Sciences*, Vol. 17, No. 3 (1978), pp. 348–375.

[11] Reiter, R. : A Theory of Diagnosis from First Principles, *Artificial Intelligence*, Vol. 32 (1987), pp. 57–95.

[12] 高橋直久, 小野諭 : 関数型プログラムの宣言的デバッグシステム DDS, 電子情報通信学会論文誌, D-I, Vol. J72-D-I, No. 11 (1989), pp. 779–788.

[13] 天満隆夫, 佐藤康臣, 森本康彦, 田中稔, 市川忠男 : プログラムの変更とその支援 – 型の変更に対する自動修正, 電子情報通信学会論文誌, D-I, Vol. J73-D-I, No. 5 (1990), pp. 510–520.

[14] 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.

[15] 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.

[16] Wand, M. : Finding the Source of Type Errors, In *Proc. 13th ACM Symp. on Principles of Programming Languages*, ACM, 1986, pp. 38–43.