1998 年度 修士論文

並行論理プログラム自動修正系 *Kima* の設計と実装

提出日: 1999 年 2 月 5 日

指導: 上田和紀 教授

早稲田大学大学院 理工学研究科 情報科学専攻

学籍番号: 697p003-9

網代 育大

要旨

プログラムの動作や性質を静的に解析する体系の下で、プログラム中の簡単な誤りを自動的に修正することのできるシステム *Kima* の実装を行なった。 *Kima* は、 KL1 プログラムの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検出・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は、プログラム実行時に発生する通信のプロトコルの一貫性を静的に保証し、プログラムの最適化などに有用であるばかりでなく、プログラムの自動デバッグにも有効である。このとき用いられるモード解析は、多数の簡単なモード制約の制約充足問題であり、素性グラフの単一化問題として効率良く解くことができる。プログラム中に誤りが存在する場合、そのプログラムから得られるモード制約の集合が充足不可能となることが多く、その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、誤りの存在する場所を局所的に特定する技術が、すでに提案されている。型体系に関しても同様のことが言え、通信ストリームを流れるデータのデータ型に関する一貫性を保証し、プログラムの自動デバッグに利用することができる。

これら既存の技術を応用することで、ユーザによってプログラムの仕様やあらゆる宣言を与えることなしに、プログラム中の誤りがどの程度、自動的に修正可能かに関する調査・研究を行なった。その結果、誤りが軽微であれば、それに対して求まる修正案も、意図通りの修正を含むごく小さい数に抑えられることが分かった。本論文では、この自動修正技術の枠組に関して詳しく述べるとともに、この枠組を利用したシステム、 *Kima* の実装を行ない、いくつかの実験データを示して、提案した枠組に対する評価を行なう。

もくじ

1 背景と目的 **4**

2 並行論理プログラミングにおける強モード / 型体系 **7**

3 静的解析による誤り箇所の同定 **12**

4 静的解析によるプログラムの自動修正 **16**

4.1 自動修正アルゴリズム 17

4.2 モード制約と型制約の併用 18

4.2.1 *Kima* の採用する型体系 19

4.3 ヒューリスティクスによる修正案への優先度の付加 20

4.4 誤り箇所のグループ化 23

4.5 アルゴリズムの適用例 24

4.5.1 例 1 — `Append` 24

4.5.2 例 2 — `Fibonacci sequence` 26

4.5.3 例 3 — `Quicksort` 28

5 *Kima* の構成と戦略 **30**

5.1 MIS の計算 32

5.2 MIS のグループ化 32

5.3 修正案の generate & test とクイックチェック 34

5.4 修正案への優先度づけ 35

6 おわりに **36**

6.1 関連研究 36

6.2 まとめと今後の課題 37

図一覧

2.1 節 h :- G | B が課するモード制約 [8] 10
2.2 クイックソートプログラムのモードグラフ 11
2.3 節 h :- G | B が課する型制約 11

3.1 バグ箇所の特定 13

4.1 修正案探索の様子と、モードと型の併用による解空間の縮小 19
4.2 グループ化したモードおよび型制約の矛盾する極小部分集合 23
4.3 修正案探索の例 25

5.1 *Kima* の構成 31
5.2 MIS 間の節の共有をポインタで表した木構造 33

表一覧

4.1 $Kima$ における型の分類 20
4.2 変数の 1 箇所の書き間違いに対する修正案の数 22

第 1 章

背景と目的

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 [3] 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良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 (constraint satisfaction problem) 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 [8]。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きた [2]。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。型解析をプログラムの診断問題 [4] としてとらえる研究はこれまであまり見

られなかったが、同様の技法は関数型言語などの型解析にも利用できるはずである。

このような制約の矛盾は、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冗長性を持っていることから起こる。現実にある多くのプログラムでは、

- 条件分岐や 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、一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、一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。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可能である [12]。

この冗長性を利用することで、たとえ誤ったプログラムであっても、モードや型などの抽象情報に関する正しい仕様を推定し、発見された軽微な誤りを自動的に修正することが考えられる。

符号理論分野における自動誤り訂正の技術と比較すると、プログラムそれ自体は誤り訂正符号のような直接的な冗長性を含んでいないものの、そこから得られるモードや型制約の集合には冗長性が存在することが多い。*Kima* は、これを利用することで、プログラムの自動修正を実現している。

自動修正は基本的に、修正案の生成とそのそれぞれに対するモード解析という generate & test 方式で行なわれるが、先ほど述べた、モード / 型制約の矛盾する極小部分集合から、誤り箇所を一定の領域に絞り込むことで、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。このフレームワークに関して言えば、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、探索空間の縮小および修正案の品質の向上その他、多くの場面で非常に有効に作用する。

今回の研究では、並行論理型言語である KL1 および、その元となっている Moded Flat GHC を例にとり、 Moded Flat GHC に備わるモード / 型体系を利用することで、この自動修正に関する有望な結果を得ることができた。ただし、ここで述べている手法の有用性は、プログラミング言語とその静的解析系がどのようなものであるかに強く依存しており、このフレームワークの他言語への適用可能性の調査については今後の課題である。

本論文では、つづく第 2 章で、この自動修正技術の基礎となっている、並行論理プログラムにおける強モード / 型体系について述べた後、 3 章では、すでに提案さ

れている、静的解析による誤りの自動検出・同定技法について簡単に説明する。そして第 4 章で、この研究の核となる、並行論理プログラムの自動修正技術の枠組について詳しく述べた後、この枠組が $Kima$ ではどのような戦略で実装されているかについて、第 5 章を使って説明する。第 6 章はそれらのまとめである。

第 2 章

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 の強モード体系とモード解析について簡単に説明する。これらはすでに提案された既存の技術であり、詳細については [8][9] 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、ちょうど一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、項と原子論理式 (ゴールや節頭部) のパスの集合は、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、 $Fun/Atom/Term$ は関数記号 / アトム / 項の集合、 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。

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 : $P_{Atom} \to \{in, out\}$ 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 *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 2.1 のようにまとめることができる。ここで Var は変数記号の集合を、 $\tilde{a}(p)$ は原子論理式 a 中のパス p に出現する記号を表わす。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もつことを許すためである。

例として次のようなクイックソートプログラムを考える:

```
quicksort(Xs,Ys):- true | qsort(Xs,Ys,[]).
qsort([],     Ys0,Ys ):- true | Ys=₁Ys0.
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).
```

全体からは 53 個の制約が得られる。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) をもつことができ、
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 2.2 が、クイックソートプログラムから得られた制約を表わすモードグラフである。ただしこの図では、 q と p がそれぞれ qsort、 part を表しており、グラフの簡潔さのために、トップレベルノードとユニフィケーションゴールは省略してある。

モード解析は、基本的には、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 [8]。図 2.1 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自身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もつプログラムでも、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。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 $Func$ を，重なりをも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 2.3 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。

この章で述べてきた技術は、KL1 プログラムのモード / 型を解析する静的解析系 *klint* として既に実装されている [10]。

(HF) $\forall p \in P_{Atom}(\widetilde{h}(p) \in \mathit{Fun} \Rightarrow m(p) = \mathit{in})$

(もし h 中のパス p に関数記号が現れるならば $m(p) = \mathit{in}$),

(HV) $\forall p \in P_{Atom}(\widetilde{h}(p) \in \mathit{Var} \wedge \exists p' \neq p(\widetilde{h}(p) = \widetilde{h}(p')) \Rightarrow m/p = \mathit{IN})$

(もし h 中のパス p に現れるのが、 h 中に複数回出現する変数であるならば $m/p = \mathit{IN}$),

(GV) $\forall p, p' \in P_{Atom}\, \forall a \in G(\widetilde{h}(p) \in \mathit{Var} \wedge \widetilde{h}(p) = \widetilde{a}(p') \Rightarrow \forall q \in P_{Term}(m(p'q) = \mathit{in} \Rightarrow m(pq) = \mathit{in}))$

(もし同じ変数が h 中の p と G 中の p' に出現するならば
$\forall q \in P_{Term}(m(p'q) = \mathit{in} \Rightarrow m(pq) = \mathit{in})$)

(BU) $\forall k > 0\, \forall t_1, t_2 \in \mathit{Term}((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle})$

(ボディの単一化ゴールの二つの引数は正反対のサブモードをもつ)

(BF) $\forall p \in P_{Atom} \forall a \in B(\widetilde{a}(p) \in \mathit{Fun} \Rightarrow m(p) = \mathit{in})$

(ボディ・ゴールの中のパス p に関数記号が現れるならば $m(p) = \mathit{in}$),

(BV) $v \in \mathit{Var}$ 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

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

ここで $\mathcal{R}$ は、パス間の “協調的通信” を表わす下記のような関係である:

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

図 2.1: 節 h :- G | B が課するモード制約 [8]

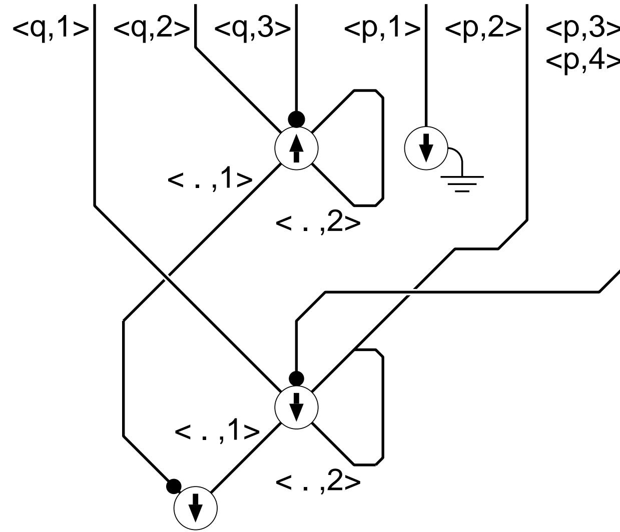

図 2.2: クイックソートプログラムのモードグラフ

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

(HBV$_\tau$) $\tau/p = \tau/p'$, 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$_\tau$) $\tau/\langle =_k, 1\rangle = \tau/\langle =_k, 2\rangle$, for a unification body goal $=_k$.

図 2.3: 節 h :- G | B が課する型制約

第 3 章

静的解析による誤り箇所の同定

ここでは、 *Kima* が利用する、静的解析によるプログラムの誤り箇所の特定技法について述べる。これは [2] によって提案されており、詳細については、 [2][12][13] を参照されたい。ここでは概略について簡単に説明する。

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。 (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 [2][1](図 3.1)。

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、同様にバグの箇所を同定することができる。ただし、モードと型は、プログラムの性質に関して、異なる領域をそれぞれ表現しており、これらを併用することで、この誤りの検出力を、より向上させることができる。また、モードと型制約の併用は、修正案の探索空間を縮小し、修正案の品質を向上させるのにも有効である。これについては、 4 章で詳しく説明する。

この矛盾する極小部分集合は、次に示すような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

[1] ここで示すアルゴリズムでは、 C が無矛盾だったときのことを考慮して、文献のものが改訂されている。

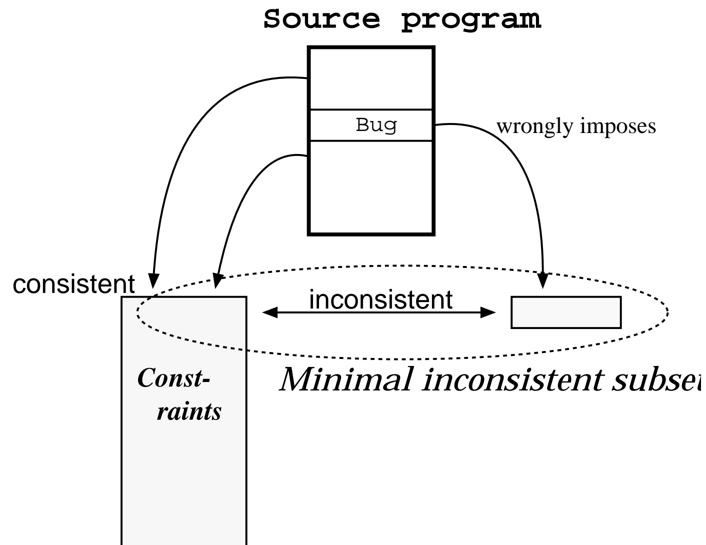

図 3.1: バグ箇所の特定

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。

アルゴリズム 1:

$c_{n+1} \leftarrow false;$

$S \leftarrow \{\};$

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** $S \leftarrow \{\}$

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、 [2] 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。

また、このアルゴリズムでは、制約の解消とその充足可能性のチェックは、第 2 章

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、その効率は制約解消系の効率に大きく依存している。

実験では、矛盾する極小部分集合の平均サイズは、おおよそ 4 である。11 より大きいモードに関する極小部分集合は、いまのところ見つかっていない。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。これは並行論理プログラムが一般に、述語の呼びだし関係に関して広く浅い構造を持っていることによっている。これは、2 章で述べた、モードグラフが広く浅い構造をしていることにも関係している。そして、この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ことによって、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、ある狭い領域に特定することができる。

この制約の矛盾は、モードや型に関する制約が冗長性を持っていることによって起こる。述語定義における条件分岐や、複数の場所からの同一述語の呼びだしによって、この冗長性が発生しているわけであるが、誤りが軽微であるという仮定から、プログラムテキストから得られる大多数の制約はプログラムの正しい仕様を表しているものとみなすことができる。そして、制約の冗長性が高い場合はさらに、

1. $n-1$ 個の無矛盾な極大集合

2. ある単一誤りに対して、いくつかの極小部分集合が考えられる場合、それらの重複部分にある制約式

を求めることで、誤りの場所を、より狭い範囲内に絞り込むことが可能である。

アルゴリズム 2 は、これら 2 つの方針による解を一度に求めるものである。計算量を削減するため、考えられる極小部分集合をすべて求めることはせず、求まった極小部分集合からある 1 つの制約式を除き、アルゴリズム 1 を適用することで、別の極小部分集合を求めている。こうすることで、方針 1 および 2 に適合する制約式を同時に求めることができる。

ここで、$S = \{s_1, \ldots, s_m\}$ をアルゴリズム 1 によって求まった極小部分集合とし、`getminimal`(C) を、さきほどのアルゴリズム 1 によって、制約のマルチ集合 C から矛盾する極小部分集合を求める手続きとする。

アルゴリズム 2:

```
T ← S;
for j ← 1 to m do
  S' ← getminimal(C \ {s_j});
  if S' = {} then
    {s_j} を方針 1 の解として出力
  else T ← T ∪̇ S';
end for
```

T は、制約のマルチ集合であり、S、S' に出現する制約式をカウントする役割をもっている。$\dot{\cup}$ は、マルチ集合の和を取る演算である。

T には、それぞれの制約式が複数の極小部分集合中に何回出現したかが記録される。方針 2 に従うと、T 中の制約式のうち、そのカウントされた出現数が多い制約ほど、矛盾の原因として疑わしいということになる。

アルゴリズム 2 は、複数の誤りを一度に検出する際にも有効である。つまり、アルゴリズム 2 によって誤りの可能性のある制約集合をより小さく求めることができれば、制約式全体から除く制約を少なくすることができ、これによって複数個の誤りの検出力を高めることができる。

方針 1 による唯一の制約式が求まる場合は方針 2 による解は必要ないが、方針 1 が解を見つけられない場合でも、方針 2 が疑わしいいくつかの制約を求められる場合が存在する。

また、アルゴリズム 2 はつねに S を、より洗練できるわけではない。このときは、S に含まれるすべての制約が、方針 1 の解として求まってしまう。しかし、S の大きさが小さいので、これはそれほど深刻な問題とはならない。

第 4 章

静的解析によるプログラムの自動修正

ここでは、第 2 章の強モード / 型体系および、3 章の誤り箇所の特定技法を利用することで、*Kima* がどのようにしてプログラムの自動修正を実現しているかについて、詳しく説明する。

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 2.1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。言い換えれば、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のである。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、プログラム中の誤りによって発生している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。

そして修正の対象とする誤りは、抽象構文木 (abstract syntax tree) の終端記号の誤りに限定して考えることにする。これはつまり、「変数や定数の少数個の書き誤り[1]」である。これは非常に限定的に聞こえるが、実際、論理型言語ではプログラム

[1]現在 *Kima* では、定数記号の修正については未実装となっている。

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。

自動修正のためのアルゴリズムは、間違ったプログラムを探索木の根とし、変数や定数を書き換えていって制約全体が充足可能になるかどうかを調べる探索アルゴリズムとなる。これは、正しい仕様と思われる制約 B と図 2.1に示すような「A ならば B」の形の制約規則から、正しいプログラムの形式 A を求める発見的推論過程と考えることもできる。

例えば、変数の 1 箇所の書き間違いを修正したい場合、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ついて考慮しなければならず、それらすべてに対して、モード / 型解析を行なって *well-moded/typed* の成否をテストしなければならない。しかし、誤り箇所をあらかじめ特定できることで、特定された記号 (出現) を中心に修正案を探索して、探索空間の縮小と計算時間の節約をはかることができる。

このとき、誤りが軽微であるという仮定から、書き換え個数に関する深さ漸増探索によって修正案の探索を行なう。ただし、書き間違いの数が 2 つ以上ある場合、そのうちの 1 つは、矛盾する極小部分集合によって必ず指摘されるものの、それ以外の記号については、極小部分集合によって必ず指摘されるという保証があるわけではない。理論的には、2 つ目以降の誤りは、極小部分集合に含まれない、プログラム全体の領域中に存在している可能性がある。しかし、この誤りに対する修正案を完全に探索するのは、極小部分集合をもとに誤り箇所を特定し、それに基づいて修正案を探索するという本技法の枠組から外れており、計算時間および探索空間のの極端な増大を招く。そこで *Kima* は、近似的な解決策として、矛盾する極小部分集合によって指摘された「(複数の) 節」の中に、2 つ目以降の誤りも存在すると仮定して、修正案の探索を行なうことにしている。

4.1 自動修正アルゴリズム

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、自動的にその修正案を求めるためのアルゴリズムは、次のようなものである。

自動修正の基本アルゴリズム:

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記号を抽出;
$depth \leftarrow 1$;
while 解が見つかっていない **do**
  while *depth* 個の記号の書換え方がまだある **do**
    その *depth* 個の記号を書換える;
    if 書換えたプログラムが *well-moded* かつ *well-typed* になった
     then その書換えを修正解として出力
  end while;
  $depth \leftarrow depth + 1$
end while;

このアルゴリズムは、モード制約の矛盾する極小部分集合から疑わしい記号を抽出する誤り同定部分と、それらをもとに修正案を探索する部分の 2 つに大別することができる。また修正案の探索は、書換え個数に関する深さ漸増探索 (iterative-deepening) であり、 *depth* がその探索の深さを表している。

4.2 モード制約と型制約の併用

アルゴリズムでは、モード制約と型制約それぞれに関して、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、プログラムのもつ異なる性質を表現しており、検出できる誤りの種類が違うからである。つまり、モードと型を併用することで、誤りの検出力を向上させ、さらに、ある単一誤りを考えた場合、モードに関する矛盾する極小部分集合と型に関するそれとでは、怪しいとして指摘する節や記号が異なることも多いため、誤りの可能性のある場所を、より正確に特定するのに役立てることができる。

別ないい方をすれば、ユーザの作り得るすべてのプログラムを要素として持つ、プログラム全体の空間というものを考えたとき、 *well-moded* であるものの集合と、*well-typed* であるものの集合は、共通部分を持った異なる集合である。そして、軽微な誤りをもつプログラムの構文的な近傍にある「ユーザの意図通りのプログラム」は、この共通部分の部分集合になっている。

well-moded であるプログラムの集合と *well-typed* であるプログラムの集合は、ともに全体集合と比べて小さいが、これらを併用することで、探索によって求まる修

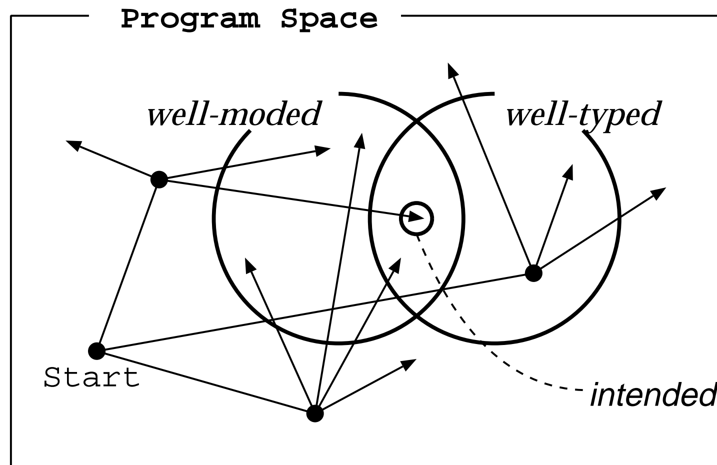

図 4.1: 修正案探索の様子と、モードと型の併用による解空間の縮小

正案の数をより減少させ、修正案の品質を向上させることができる (図 4.1)。

表 4.2 は、プログラム中の変数の書き間違いを網羅的につくり、それに対してモードだけを利用した場合と、型を併用した場合の、誤りの検出数の増加を調べたものである。 append プログラムでは、データ型としてリスト型しか利用していないために、型を併用しても検出率が向上していないものの、全体としては検出率が 58.0% (256/441) から 70.5% (311/441) に向上している。

append のように型が 1 種類しか現れないプログラムを考えた場合、型制約だけでは変数の書き間違いのような誤りは検出することができない。この点で並行論理型言語におけるモード体系の役割は大きいということができる。しかし複数の型が使われないようなプログラムの場合でも、型解析の併用は、修正案における、書換え先となる定数のデータ型を推定するときなどに有用である。

4.2.1 *Kima* の採用する型体系

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、整数、浮動小数、文字列、ベクタ、リスト、ストラクチャの 6 つの型に分類し、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 章)。また、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。これらをまとめたものを、表 4.1 に示す。ただし、ストラクチャ型における atom(*Atom*) の *Atom* は、‘[]’ 以外の記号アトムを表している。

表 4.1: *Kima* における型の分類

型	関数記号	KLIC 処理系における *wrapped term*
F_1	整数	integer(*Int*)
F_2	浮動小数	floating_point(*Float*)
F_3	文字列	string(*Str*)
F_4	ベクタ	vector({*Elem, ...*})
F_5	リスト	list([*Car*| *Cdr*]) または atom([])
F_6	ストラクチャ	functor(*Functor*(*Arg*, ...)) または atom(*Atom*)

4.3 ヒューリスティクスによる修正案への優先度の付加

Kima はさらに、求めた修正案に対し、以下のようなヒューリスティクスを使って優先度をつける。

1. モード制約の強さの総和が弱い方が修正案としてもっともらしい。
2. データ型を表した型グラフにおいて、リストの car と元のリストのデータ型が単一化されているものは、誤りである可能性が高い。
3. ガードにある変数がヘッドにない場合は、誤りである可能性が非常に高い。
4. 単一化述語のの両側に同一の変数が出現している場合は、誤りである可能性が非常に高い (簡単な occur-check[2])。

ヒューリスティクス 1 は主に、

1-1 ボディにおける変数の単独出現 (singleton variable)、

1-2 ヘッドにおける同一変数の 2 回以上の出現

があるようなプログラムを「もっともらしくない」プログラムとして検出するためのものである。まず、1-1 のような変数がパス p にある場合、図 2.1 のモードづけ規則 (BV) から、$m/p = OUT$ に制約される。これは、その変数がある手続きによって具体化されるものの、そのデータがどこからも参照されていないということを意味しており、このような変数が存在しない修正案よりも「もっともらしくない」こ

[2]完全な occur-check については未実装である

とになる。また 1-2 は、同じくモードづけ規則 (HV) から、そのような変数がヘッドのパス p_1, p_2, ... に存在する場合、これらは $m/p_i = IN$ に制約され、それらの変数の具体値が完全に等しいか否かが実行時にチェックされる。このような修正案は、それがない場合よりも、実行時にリダクションの失敗を起こすことが多い。これらの経験から、ヒューリスティクス 1 が定められている。

ヒューリスティクス 2 は例えば、4.5.1 における修正案 5 のようなものを「もっともらしくない」とみなす。この修正案では、変数 A が、リスト型に制約されているパス $\langle \mathrm{append}, 1\rangle$ と、パス $\langle \mathrm{append}, 1\rangle\langle\, .\,, 1\rangle$ (つまりリストの car) とに同時に出現している。このようなプログラム節は必ずしも誤りではないが、この「リストの要素のデータ型がまたリストになっている」ような場合、実行時にこの節が選ばれてリダクションを起こすたびに、リストのデータ構造が深くなっていくことになる。このようなものも経験的に誤りである可能性が高い。

ヒューリスティクス 3 は、ガードで判定すべきデータが、その述語の呼出し時に存在しないことを意味しており、明らかに誤ったプログラムである。ヒューリスティクス 4 も、変数の出現検査 (occur-check) の観点から、誤りである可能性が非常に高い。

表 4.2 は、プログラム中の変数の書き間違いを網羅的につくり、前述のヒューリスティクスを使って、最も優先度の高いものだけを修正解として求めた場合と、ヒューリスティクスを利用しなかった場合の、提示される修正案の数の比較である。ただし、ガード変数の書き誤りに関しては、カウントをしていない。また、ヒューリスティクス 3 に当てはまるような修正案は明らかに誤りであると考え、「ヒューリスティクスなし」の時点ですでに除いてある。さらに、型を併用したことによって新たに検出された誤りに関してだけ、リストの cdr に出現する変数にリスト型の制約が課せられて (5.1節参照) いることによって、提示される修正案数が減っている可能性があるが、それ以外の誤りについては、これはなされていないので、データを読む際は注意されたい。

表 4.2: 変数の 1 箇所の書き間違いに対する修正案の数

プログラム	利用情報	ヒューリスティクス	実験総数	検出成功	求まった修正案の数							
					1	2	3	4	5	6	7	≥8
append	モードのみ	–	58	36	–	–	–	–	–	–	–	–
	モード & 型	あり	–	36	20	14	2	0	0	0	0	0
		なし	–	–	1	3	7	3	8	4	3	7
fibonacci	モードのみ	–	104	51	–	–	–	–	–	–	–	–
	モード & 型	あり	–	66	53	4	2	5	1	1	0	0
		なし	–	–	25	16	3	9	8	1	2	2
quicksort	モードのみ	–	279	169	–	–	–	–	–	–	–	–
	モード & 型	あり	–	209	146	63	0	0	0	0	0	0
		なし	–	–	74	79	2	28	1	7	8	10

発表論文 1 における類似のデータとの違いについて

論文 1 における実験では主に

- ガードの変数の書き間違いの他に、ヘッドの変数を書き間違えたことによってヒューリスティクス 3 のような状況になってしまうような間違いについても、カウントを行なっていない
- 修正案数については、 mode のみを利用した場合では ヒューリスティクス 1-1 だけを利用しており、 type を併用したときにはヒューリスティクス 2 も使って修正案を絞り込んでいる
- 型を併用した際、リストの cdr が変数である場合、その変数にリスト型の制約がかけられていない
- type を併用したことによって新たに見つかった誤りに対しては、修正案を求めていない

ために、データの値が違ってきている。その他、 *Kima* のバージョンの違いも影響しているので、比較の際は注意して欲しい。

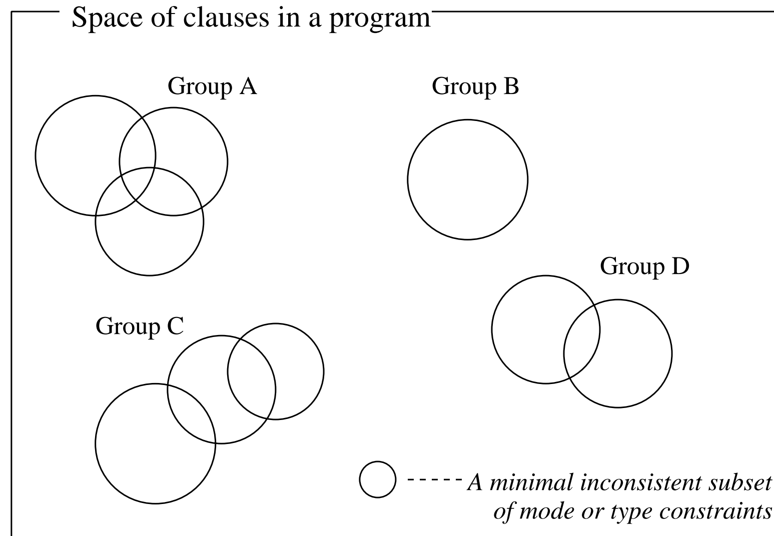

図 4.2: グループ化したモードおよび型制約の矛盾する極小部分集合

4.4　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、独立して複数求まるが、それらに含まれている制約が指摘する節や記号は、複数の極小集合によって共有されていることも多い。特に、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ときは、モードと型両方の矛盾する極小部分集合によって、制約を課した原因である節と記号が共有されることになる。このとき、指摘している節を共有する極小部分集合同士を、誤りに対する同じ「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 4.2)。

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ないかを先にチェック (**クイックチェック**) することで、モード / 型解析によって、書換えた後のプログラムが *well-moded/typed* か否かを無駄に調べる手間を減らし、計算時間を軽減させることができる。つまり、矛盾する極小部分集合に含まれている制約は、

- その記号出現を他の記号に書換える、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 2.1のモードづけ規則 (BV)

を参照)

ことによって変化し、矛盾が解消する可能性があるが、クイックチェックとは、ある *depth* 個の書換えが、この 2 つのうちのどちらかの条件を満たすことにより、グループ内のすべての極小部分集合について、その要素である制約が変化する可能性があるかどうかを調べることである。

このグループ化を考慮した自動修正アルゴリズムは次のようになる。

グループ化を使った自動修正アルゴリズム:

(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;
　$depth \leftarrow 1$;
　while 解が見つかっていない **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を通らない **then continue**
　　　end if;
　　　その *depth* 個の記号を書換える;
　　　if 書換えたプログラムが *well-moded* かつ *well-typed* になった
　　　　then その書換えを修正解として出力
　　　end if;
　　end while;
　　$depth \leftarrow depth + 1$;
　end while
end for

4.5 アルゴリズムの適用例

4.5.1 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。

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).`
　(R_2 の頭部は `append([A|X],Y,Z0)` が正しい)

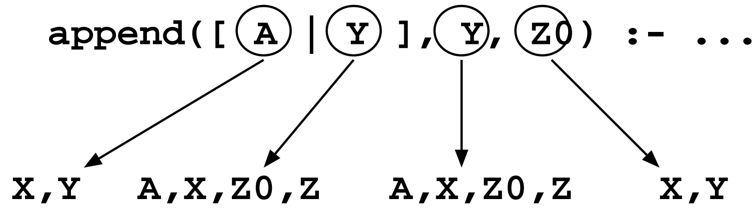

図 4.3: 修正案探索の例

このプログラムは *ill-moded* となり、3 章のアルゴリズム 1 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	原因となる記号
(a)	$m/\langle \text{append}, 1\rangle\langle .\,, 2\rangle = \mathit{IN}$	(HV)	`Y` in R_2
(b)	$m/\langle \text{append}, 1\rangle = \mathit{OUT}$	(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分かる。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、この、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、型に関してのエラーは発生しない。これは、`append` プログラムに現れるデータ型が、リスト型とその要素の 2 種類しかないからである。しかしこのような場合でも、モードに関する制約は、プログラム中の誤りを静的に検出し、誤りの原因となっている記号を絞り込むのに成功している。

自動修正アルゴリズムは、まず 1 箇所を書換えることでモード制約に関する矛盾を解消し、修正案を見つけようとする。このときの探索の様子を表したのが、図 4.3 である。探索の結果、次の 6 つの修正案が解として求まる。

(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).`

その後、修正案 (3), (4), (5), (6) は、変数 Y の頭部における複数出現が存在するために、 4.3 のヒューリスティクス 1 によって、優先度が低くなる。実際にこれらの修正案に基づくプログラムを走らせようとしても、リダクションの失敗のために、ほとんどの入力に対して、実行は失敗してしまう。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、リストをつなげる機能を持った、正しい append プログラムである。しかし残った修正案 (2) も、 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。つまり append([a,b,c],[d,e,f],Res) などとして呼び出したとき、意図通りの修正案 (1) では、 [a,b,c,d,e,f] が結果として求まるが、この修正案 (2) では、 [a,d,b,e,c,f] が結果として求まる。

6.2 で今後の課題として述べている「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も可能であるが、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うことができる。

また、この例の場合、 3 章のアルゴリズム 2 を使うことにより、怪しいとされるモード制約はさらに、 (b) ひとつに絞り込むことがでる。これにより、修正案は、変数 X の周辺についてだけ調べれば良くなり、探索空間がさらに狭められる。

4.5.2 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: R_1 : fib(Max,N1,N2,Ns0):- N2 >Max | Ns0 $=_1$ [].

2: R_2 : fib(Max,N1,N2,Ns0):- N2=<Max |

3: N1 $=_2$ [N2|Ns1], N3:=N1+N2, fib(Max,N2,N3,Ns1).

(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、それぞれについて極小部分集合が求まる。

モード制約に関しては、次のような 2 つの独立する極小部分集合が発見される。

極小部分集合 1 (モード):

	モード制約	規則	原因となる記号
(a)	$m(\langle \texttt{=}_1, 2\rangle) = \mathit{in}$	(BF)	“[]” in R_1
(b)	$m/\langle \texttt{=}_1, 1\rangle = m/\langle \texttt{fib}, 4\rangle$	(BV)	Ns0 in R_1
(c)	$m/\langle \texttt{=}_1, 2\rangle = \overline{m/\langle \texttt{=}_1, 1\rangle}$	(BU)	$\texttt{=}_1$ in R_1
(d)	$m(\langle \texttt{fib}, 4\rangle) = \mathit{IN}$	(BV)	Ns0 in R_2

極小部分集合 2 (モード):

	モード制約	規則	原因となる記号
(e)	$m(\langle \texttt{=}_2, 2\rangle) = \mathit{in}$	(BF)	“.” in R_2
(f)	$m/\langle \texttt{=}_2, 2\rangle = \overline{m/\langle \texttt{=}_2, 1\rangle}$	(BU)	$\texttt{=}_2$ in R_2
(g)	$m(\langle \texttt{=}_2, 1\rangle) = \mathit{IN}$	(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	原因となる記号
(h)	$\tau/\langle \texttt{fib}, 2\rangle = \tau/\langle \texttt{:=}, 2\rangle\langle \texttt{+}, 1\rangle$	(BV_τ)	N1 in R_2
(i)	$\tau(\langle \texttt{=}_2, 2\rangle) =$ リスト型	(BF_τ)	“.” in R_2
(j)	$\tau/\langle \texttt{fib}, 2\rangle = \tau/\langle \texttt{=}_2, 1\rangle$	(BV_τ)	N1 in R_2
(k)	$\tau/\langle \texttt{=}_2, 2\rangle = \tau/\langle \texttt{=}_2, 1\rangle$	(BU_τ)	$\texttt{=}_2$ in R_2
(l)	$\tau(\langle \texttt{:=}, 2\rangle\langle \texttt{+}, 1\rangle) =$ 整数型	builtin	:= in R_2

これら、極小部分集合 1, 2, 3 は、怪しい節としてどれも R_2 節を指摘しており、これによって同じグループに分類される。指摘されている終端記号についてまとめると次のようになる。

節	終端記号	指摘する極小部分集合
R_1	Ns0	1
	“[]”	1
R_2	Ns0	1
	N1	2, 3

そして深さ漸増探索により、まず 1 箇所の書換えによる修正案の探索が行なわれる。このとき、 R_1 節において、変数 Ns0 を他の変数に書換える、あるいは Ns0 の出現を増やすといった探索は、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、すべての極小部分集合を解消させる可能性がないために、クイックチェック

によって、そのような探索を行なわなくて良いことが即座に分かる。“[]”に関しても同様である。

R_2 節における書換えを考えた場合、1箇所の書換えによってすべての極小部分集合を解消させうるような書換え方は、Ns0 を N1 に書換えるかあるいはその逆 (N1 を Ns0 に書換える) 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 8 通りのプログラムに関してだけ、モード / 型解析を行なって、制約全体が充足するか否かを調べれば良いことになる。

この結果、次のような修正案が、ただ1つだけ求まる。

(1) Line 3: Ns0 $=_2$ [N2|Ns1], N3:=N1+N2, fib(Max,N2,N3,Ns1).

これは、ユーザの意図通りの正しい修正となっている。

4.5.3 例 3 — Quicksort

最後に、2つの変数をお互いに書き間違えているクイックソートプログラムの例を示す。

```
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).
        (Ys1 =2 [X|Ys2] が正しい)
```

アルゴリズム1は、これ対して以下のような、モード制約の矛盾する極小部分集合を求める。型に関する矛盾は発生しない。

	モード制約	規則	原因となる記号
(a)	$m(\langle \mathtt{qsort}, 3\rangle) = in$	(BF)	“[]” in R_1
(b)	$m/\langle =_1, 1\rangle = m/\langle \mathtt{qsort}, 3\rangle$	(BV)	Ys in R_2
(c)	$m/\langle =_1, 2\rangle = \overline{m/\langle =_1, 1\rangle}$	(BU)	$=_1$ in R_2
(d)	$m/\langle \mathtt{qsort}, 2\rangle = m/\langle =_1, 2\rangle$	(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)	$m/\langle =_2, 1\rangle = \overline{m/\langle \mathtt{qsort}, 2\rangle}$	(BV)	Ys2 in R_3

これらの制約からは実際、次の明らかに矛盾する 2 つの制約を導くことができる。

$$m(\langle \texttt{qsort}, 2\rangle) = out, \quad \text{by (a), (b), (c) and (d),}$$
$$m(\langle \texttt{qsort}, 2\rangle) = in, \quad \text{by (e), (f) and (g).}$$

制約の冗長性が低いために、アルゴリズム 2 を使っても、怪しい制約をこれ以上絞り込むことができない。これにより、$\langle \texttt{qsort}, 2\rangle$ やその他のパスの正しいモード値が決定できず、それらの可能性をすべて考慮する必要が発生して、探索空間が大きくなることになる。

アルゴリズムの適用により、次に示す唯一の修正案が求まる。

(i) Line 5: `Ys2=`$_2$`[X|Ys1],qsort(L,Ys1,Ys3).`

しかしこの修正案は、モードおよび型誤りは起こしていないものの、ヒューリスティクス 1-1 の、ボディに単独出現している変数が存在している。そこで、 2 箇所の記号を書換えることによる、深さ 2 の探索を行なうことにする (深さ漸増探索)。すると次に示す 5 つの修正案が、優先度の高いものとして求められる。

(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: `Ys1=`$_2$`[X|Ys2],qsort(L,Ys2,Ys3).`

ここで、 c はある定数を表している。

しかし、この `quicksort` を呼び出している側では、 `quicksort` の 1 引数目にソート前の整数値リストを入れて、その結果を 2 引数目から得るという、 $m(\langle \texttt{quicksort}, 1\rangle) = in$, $m(\langle \texttt{quicksort}, 2\rangle) = out$ となる使い方をしているわけであり、この情報が利用できれば、修正案 (1), (2), (4) は修正案から除去することができる。ここでは、修正案を求めるのに、 `quicksort` の定義部分しか使っていないために、これらの修正案がすべて提示されている。

残った修正案 (3), (5) のうち、 (5) がユーザの意図通りの整数列を昇順にソートする正しい修正案である。そして修正案 (5) は、面白いことに、 c として空のリスト ‘`[]`’ を選ぶことによって、整数列を「降順」にソートするプログラムとなる。

第 5 章

Kima の構成と戦略

ここでは、これまでに述べてきた KL1 プログラムに対する誤り自動修正のフレームワークを利用することで、 *Kima* がどのような戦略を用いて実装されているかについて説明する。

Kima は、既に存在する KL1 プログラム静的解析系 *klint* [10] をベースにして作られており、その全体構成は図 5.1のようになっている。この図におけるそれぞれの構成要素は、次のような機能を持っている。

1. **Constraints generator** モードおよび型に関する制約を生成。

2. **computing MISs on modes/types** モード / 型に関する (独立複数の) 矛盾する極小部分集合[1]を求める。

3. **grouping all MISs** 4.4 節の方針に基づき、 MIS をグループ化。

4. **Alternatives generator** グループ化した MIS に基づき、怪しいとされる (複数の) 節内の $depth$ 個の変数を書換えた修正案の集合 A を生成。

5. **Quick check** 4.4 節の「クイックチェック」による修正案集合 A の検査。

6. **mode & type analyses** クイックチェックに通った修正案集合 B に対して、モード / 型解析を行なって well-moded/typed ness を検査する。

7. **prioritizing alternatives** 最終的に求まった修正案集合 C に節 4.3 のヒューリスティクスを使って優先度をつける

ここで、修正案集合 A, B, C は、 $A \supseteq B \supseteq C$ という関係になっている。

klint は、プログラムテキストから

[1] ここでは簡単のため MIS と書く

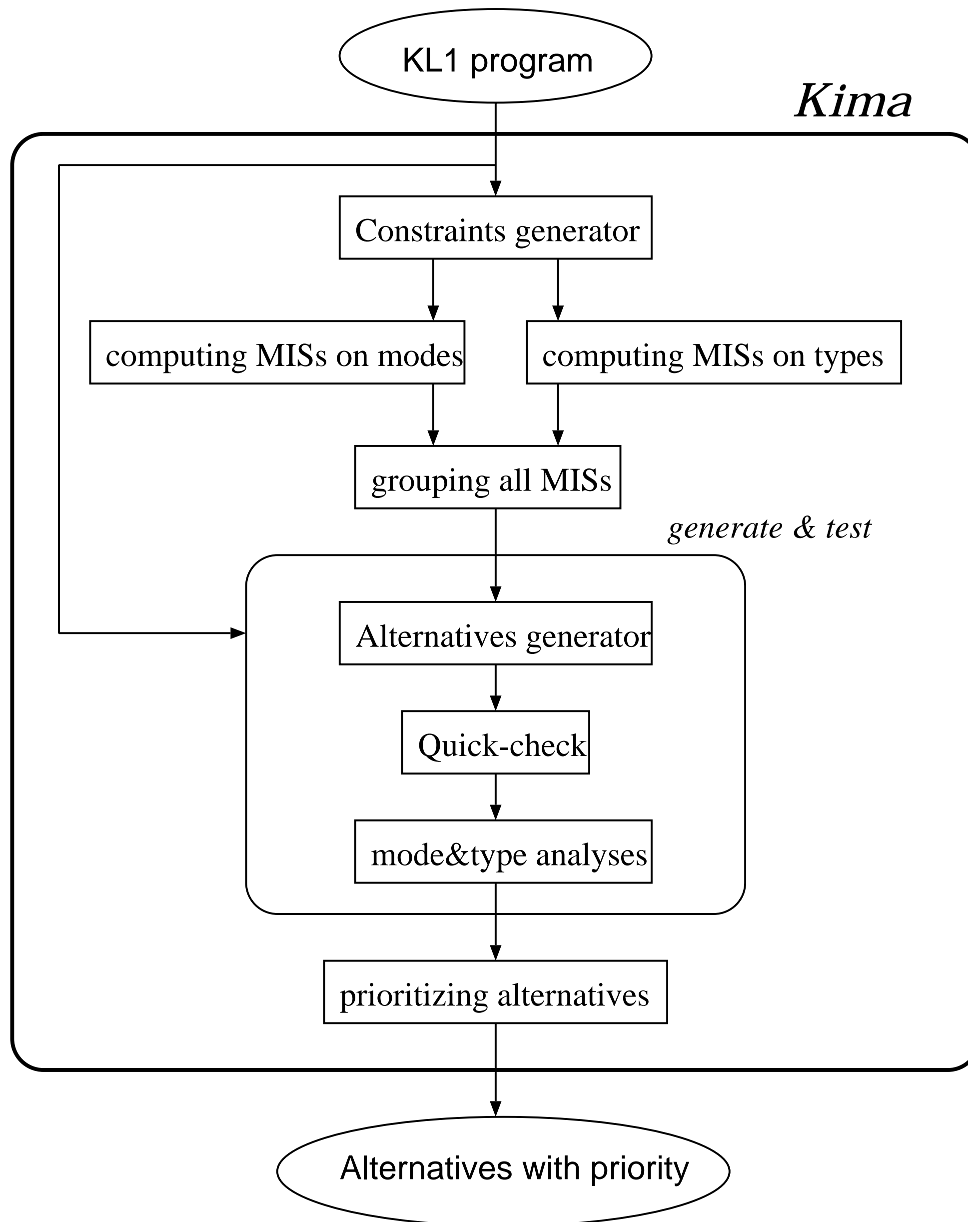

図 5.1: *Kima* の構成

- モード / 型制約を生成する機能と、
- 制約を解消する機能

を備えているため、 *Kima* の機能の 1, 6 については、これをそのまま利用することができる。その他の機能について、次からの節を使って述べていく。

5.1 MIS の計算

klint は、モードに関しては強モード体系を規定しており、制約解消系が矛盾を検出する機構を備えているものの、型に関しては強型体系を規定していない。そこで *Kima* では、 4.2.1 節で規定する型に基づいて制約生成系と解消系の一部を変更して、強型体系を規定し、型誤りを検出できるようにしている。 *Kima* の機能 1 および 6 で行なわれるモード / 型解析についても同様である。

そして *Kima* は、矛盾の発生を検出する機構を利用し、 3 章のアルゴリズムを使って、独立する複数の MIS の計算を行なう。

またその他に *klint* の行なう解析との違いとして、 *Kima* は、リストの cdr が変数である場合に、その変数に「リスト型」の型制約を課す、という処理を行なっている。これは cdr の定義からの自然な帰結であるが、 誤りの検出が目的でない *klint* では、行なわれていない。

5.2 MIS のグループ化

MIS が求まったことにより、その MIS の原因となっている節と記号が特定される。各 MIS が原因として指し示す節に重なりがあるかどうかで MIS をグループ化したい。このときの主な手続きは、独立して複数求まるそれぞれの MIS について、それらのグループ番号を求めることである。グループ番号を決定することができれば、あとはそれに基づいて、 MIS をグループ化すればよい。

Kima は次のようにして、各 MIS について、グループ番号を決定している。グループ番号は、実際には、グループ内の MIS のなかでもっとも番号の若いものの番号である。つまり、グループ番号はそれぞれのグループに対して 1 から順につけられるとは限らない。

1. すべての MIS に一意な番号 (MIS 番号) をつけ、各制約とそれを課す節と記号の対を、 MIS 番号とともに、いったんバラす。

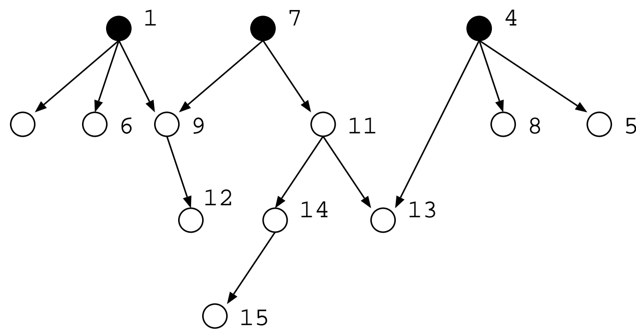

図 5.2: MIS 間の節の共有をポインタで表した木構造

2. それらを、節の名前 (内部的な表現は '`place(Module,Predicate,Nth of clauses)`') を文字列とみなしてソートする。

3. 同一節を指している異なる MIS がある場合、 MIS 番号同士の関係づけを行なう。このとき、 MIS 番号の小さいものが大きいものを指すポインタを作成する。

4. 作成したポインタを元に、それぞれの MIS についてグループ番号を決定する。作成されたポインタは、一般には図 5.2 のような木構造となっており、グループ番号は以下のようにして求められる。

   (a) まずそれぞれの MIS に対して所属番号を与える。初期状態では、みな自分自身に所属させる。

   (b) MIS 番号の若い方から順に、まず自分を含む子ノードすべてを traverse してそれらの所属番号のリストを得る。

   (c) それらの最小値が (とりあえずの) グループ番号であるので、この値をすべての子ノードに broadcast して、所属グループを更新する。この (a), (b), (c) の手続きを、まだ自分自身に属しているすべてのノードについて行なう。

1, 2 を行なうのは、 3 でポインタを作成するための手間を抑えるためである。ソートを行なわずにポインタを作成する場合、すべての MIS に含まれる制約の数の合計を n としたとき、つねに $O(n^2)$ の手間がかかることになるが、クイックソートなどを使うことで、この手間を $O(n \log n)$ にすることができる。

図 5.2 におけるすべてのノードのグループ番号は 1 になる。ノードはそれぞれの MIS に対応しており、ポインタひとつひとつが節の共有を表している。 MIS 番号 1

と 4 の処理が終っており、 MIS 番号 7 とその子ノードについてのグループ番号を求めるようなシチュエーションを考えた場合、 (a), (b), (c) の手続きが必要であることがわかる。

この手続きには、木構造の深さの平均を d、 MIS の個数を m としたとき、 $O(d \cdot m)$ の手間が必要であり、これは最悪 $O(m^2)$ となるが、現実の KL1 プログラムでは、このように MIS 同士が複雑に重なって検出されることはほとんどなく、この手間が問題になることはない。

グループ化の結果として得られるデータは、

i) そのグループに属するすべての MIS 番号のリスト

ii) グループ内の制約を課している節と節における変数記号

iii) ii に対応する (複数の) MIS の MIS 番号

である。i は、クイックチェックのために使用される。

このグループ化に基づき、プログラム中の節を、疑われている節と、疑われていない節に分け、疑われている節については、節中の記号を書換えた修正案を生成する。

5.3 修正案の generate & test とクイックチェック

それぞれのグループに対し、深さ (書換え個数) d の修正案の探索を行なう場合、まず以下のような方法で、グループ内の (疑わしいとされている) 節に関する修正案を網羅的に生成する。

1. グループ内の節に d を振り分ける。節の数を l とすると、このときの振り分け方は、重複組合せを使って、 ${}_l\mathrm{H}_d$ で計算される。

2. ある節に割り当てられた書換え数が n であるとき、その節内の変数記号の出現数を r とすると、どの記号を書換えるかの組合せは ${}_n\mathrm{C}_r$ で計算される。

3. ある変数記号を書換えるとき、その節に存在する変数の種類数が k であるすると、その書換え方としては、自分以外の変数か、あるいは、まったく新しい変数への書換えが考えられ、計 $(k-1)+1=k$ 通りの書換え方が存在する。

4. 上のすべてに関する「場合の積」を考え、節を書換えたものを修正案として生成する。

それぞれの修正案について、増加あるいは減少した変数記号に対応する MIS 番号を、前節の iii を使って求め、それらの合計がグループ内のすべての MIS (前節の i) を含んでいるかどうかを調べることで、クイックチェックを行なう。クイックチェックに通った修正案については、モード / 型解析を行なって、疑われていない節から得られる制約集合との充足可能性を調べ、最終的な修正案を求める。

ちなみに、ここで利用している重複組合せ H と組合せ C は、次のような再帰的定義を使って簡単に定義することができる。

$$
{}_n\mathrm{H}_r = \begin{cases} \sum_{k=0}^{r} {}_{n-1}\mathrm{H}_{r-k} & (r \neq 0, n \neq 1) \\ 1 & (r = 0) \\ 1 & (n = 1) \end{cases}
$$

$$
{}_n\mathrm{C}_r = \begin{cases} {}_{n-1}\mathrm{C}_{r-1} + {}_{n-1}\mathrm{C}_r & (n > r > 0) \\ 1 & (n = r) \\ 1 & (r = 0) \end{cases}
$$

5.4 修正案への優先度づけ

well-moded/typed となった修正案に対し、4.3 節で述べたヒューリスティクスを使って、優先度をつける。ヒューリスティクスをここに再掲する。

1. モード制約の強さの総和が弱い方が修正案としてもっともらしい。
2. データ型を表した型グラフにおいて、リストの car と元のリストのデータ型が単一化されているものは、誤りである可能性が高い。
3. ガードにある変数がヘッドにない場合は、誤りである可能性が非常に高い。
4. 単一化述語のの両側に同一の変数が出現している場合は、誤りである可能性が非常に高い (簡単な occur-check)。

Kima は、これらに当てはまるものについて、ペナルティを課していき、最終的なペナルティポイントによって、修正案に優先度をつける。ヒューリスティクス 1 では、モードグラフにおけるグラウンドされたノードの数をそのままペナルティポイントとし、ヒューリスティクス 2 は、型グラフにおいて car の素性 (アーク) がアークの親ノードを指しているかどうかをチェックすることで検査できる。またヒューリスティクス 3, 4 は、静的解析時の変数出現テーブル (制約生成系が作成している) を利用することで、簡単に検査することができ、これに当てはまるような修正案については、修正案から完全に排除してしまう。

第 6 章

おわりに

6.1 関連研究

これまでの (並行) 論理型言語のモード解析手法は、ほとんどが抽象解釈技術に基づくものであり、与えられたプログラムは正しいものと仮定して、その性質を推論していた。これに対して制約ベースのモード解析では、制約が無矛盾であることを正しいプログラムの必要条件と見なすことによって、静的解析を、最適化ばかりでなくプログラムの静的診断にも役立てることができる。

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 [4]、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。本論文の枠組は、デバッグ対象のプログラムの本来の仕様を陽に与えることなく、誤りの箇所を特定することを目的としている点が異なる。

関数型プログラムにおける型誤りの診断技法は [11] 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。さらに自動修正への応用は、本研究の独創的な点であるといえる。

6.2 まとめと今後の課題

強モード / 型体系の下で、ユーザによってプログラムに関する宣言を与えることなしに、プログラムの自動デバッグがどの程度可能かについて調査した。その結果、多くの場合、KL1(Moded Flat GHC) プログラムの簡単な誤りに対して、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分かった。

ユーザによる宣言が利用できる場合、それを正しい制約とみなすことで、より小さな矛盾する極小集合を得ることができ、修正案の探索空間を狭めることができる。しかしそのような宣言がなくても、強モード / 型体系と静的解析は、自動修正に関して十分強力だと言うことができる。

これは、プログラム構造の近さに関する距離空間のようなものを考えたとき、*well-moded* かつ *well-typed* である正しいプログラムの近傍には、*well-moded* かつ *well-typed* なプログラムが少数しか存在しないと予想されるからである。この、全プログラム空間における、*well-moded/typed* なプログラムの分布構造が新たな興味の対象となっている。

また今回は、プログラムを木構造に見立てた場合の終端記号である、変数や定数の書き間違いを修正の対象としたが、型情報の利用によって、単一化と代入の述語記号 (‘=’ と ‘:=’) の誤りなどの非終端記号の自動修正も現実的なものとなる。このような修正技術そのものの拡張と、型つき関数型言語などへの、本フレームワークの適用可能性の調査がさらに必要と思われる。

さらに、本フレームワークと帰納論理プログラミング (Inductive Logic Programming) との関係も興味のあるところである。帰納論理プログラミングは、プログラムに対する入力と出力を与えることによって、プログラムの正しい形を推論するものであるが、この「プログラムへの入力と出力」は、本フレームワークにとっても有用である。と同時に、本フレームワークが、帰納論理プログラミングにおける推論に利用できると思われ、この関連性の調査も今後の課題の一つである。

実装面では、定数記号の修正、完全な occur-check の他、プログラムの層化 (stratification)[2] への対応が未実装となっている。特に occur-check については、主に解析効率に関して不明な点がまだ多く、さらなる調査が必要である。

謝辞

NTT 基礎研究所の小川瑞史氏には、関連研究への貴重なコメントを頂きました。また、上田研究室の加藤紀夫君には、第 3章のアルゴリズムに対する鋭い御指摘を頂きました。そして、研究内容はもちろんのこと、研究の方法論などに関する様々な御指導を下さった上田教授、またいつも有形無形の助力を頂いている研究室内の皆様、これらの方々に、この場を借りて深く感謝致します。

論文発表および受賞

1. Ajiro, Y., Ueda, K., 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.
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. 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. ソフトウェア科学会第 14 回大会論文集, 1997, pp. 533–536.
4. 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. 人工知能学会第 11 回全国大会論文集, 1997, pp. 205–208.
5. 平成 9 年度 KLIC プログラミング・コンテスト並列環境部門佳作入賞.

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] 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.

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

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

[5] Somogyi, Z., Henderson, F. and Conway, T., The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *J. Logic Programming*, Vol. 29, No. 1–3 (1996), pp. 17–64.

[6] Ueda, K., I/O Mode Analysis in Concurrent Logic Programming. In *Proc. Int. Workshop on Theory and Practice of Parallel Programming*, LNCS 907, Springer, 1995, pp. 356–368.

[7] Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming (ICLP'90)*, The MIT Press, 1990, pp. 3–17.

[8] 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.

[9] 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.

[10] Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H9-index-E.html`, 1998.

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

[12] 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.

[13] 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.