Kima: an Automated Error Correction System for Concurrent Logic Programs [†]

Yasuhiro Ajiro and Kazunori Ueda
({ajiro,ueda}@ueda.info.waseda.ac.jp)
Department of Information and Computer Science, Waseda University

Abstract. We have implemented Kima, an automated error correction system for concurrent logic programs. Kima corrects near-misses such as wrong variable occurrences in the absence of explicit declarations of program properties.

Strong moding/typing and constraint-based analysis are turning out to play fundamental roles in debugging concurrent logic programs as well as in establishing the consistency of communication protocols and data types. Mode/type analysis of Moded Flat GHC is a constraint satisfaction problem with many simple mode/type constraints, and can be solved efficiently. We proposed a simple and efficient technique which, given a non-well-moded/typed program, diagnoses the "reasons" of inconsistency by finding minimal inconsistent subsets of mode/type constraints. Since each constraint keeps track of the symbol occurrence in the program, a minimal subset also tells possible sources of program errors.

Kima realizes automated correction by replacing symbol occurrences around the possible sources and recalculating modes and types of the rewritten programs systematically. As long as bugs are near-misses, Kima proposes a rather small number of alternatives that include an intended program. Search space is kept small because the minimal subset confines possible sources of errors in advance. This paper presents the basic algorithm and various optimization techniques implemented in Kima, and then discusses its effectiveness based on quantitative experiments.

Keywords: Concurrent Logic Programming, Types, Static Analysis, Debugging, Constraint Satisfaction

1. Introduction

In our previous work (Ajiro et al., 1998), we proposed a framework of automated debugging of program errors based on static, constraint-based program analysis in the absence of programmers' declarations. The framework was then implemented in Kima, an automated error correction system for concurrent logic programs, which featured several improvements to make the system more practical and efficient.

The mechanism of error correction in Kima is based on the mode and type systems of Moded Flat GHC (Ueda and Morita, 1990; Ueda

[†] An earlier version of this article (written in Japanese) appeared in *Computer Software* published by Japan Society for Software Science and Technology (Ajiro and Ueda, 2001). In the current version, major revision has been made on Sections 4, 5, and 6.

and Morita, 1994). 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. Concurrent logic languages (Ueda, 1999) provide a simple and powerful model of concurrency as well as a full-fledged programming language with

- first-class message channels,
- evolving process structures and channel mobility
- data structures such as lists and arrays, and
- messages with reply boxes.

All these features are due to the power of logical, single-assignment variables. Concurrent processes communicate with each other using shared logical variables. Because a logical variable can be written (or instantiated) only once, repeated message passing is realized by instantiating a shared variable to a stream (implemented as a list) of messages incrementally from the first element downwards. When a message has as its argument a reply box, which is another logical variable, that variable is instantiated by the receiver of the stream. In this case, the whole data structure of a stream is determined cooperatively by both the sender and the receiver of the stream.

Languages equipped with strong typing enable the detection of type errors by checking or reconstructing types. The best-known framework for type reconstruction is the Hindley-Milner type system (Milner, 1978), which allows us to solve a set of type constraints obtained from program text efficiently as a unification problem.

Similarly, the mode system[1] of Moded Flat GHC allows us to solve a set of mode constraints obtained from program text as a constraint satisfaction problem. Mode reconstruction statically determines the read/write capabilities of variable occurrences and establishes the consistency of communication protocols between concurrent processes (Ueda and Morita, 1994). In other words, mode reconstruction guarantees the cooperative use of shared variables between concurrent processes. By cooperative we mean exactly one process can determine each part of a data structure (such as a stream of messages) communicated between processes.

As we will see later, types in Moded Flat GHC also can be reconstructed using a similar (and simpler) technique. Further details of our mode and type systems are found in Sect. 2.

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

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

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 (Cho and Ueda, 1996). 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 mode errors.

Using the information of possible locations of bugs, automated correction is attempted basically by generate-and-test search, namely the generation of possible rewritings and the computation of their principal modes and types. Search space is kept small because the locations of bugs have been limited to small regions of program text.

A significant feature of our framework is that it works on a fragment of a program such as a set of predicate definitions in a particular module. Our framework is quite effective, for example, when a program has not been completely constructed. This is due to the fact that the multiset of mode constraints imposed by a program usually has redundancy for 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.

Although the framework is quite general, whether it is practical or not may depend on the choice of a language. Kima corrects wrong occurrences of variable symbols in a KL1 (Ueda and Chikayama, 1990) program assuming strong moding and typing of Moded Flat GHC. KL1 is designed based on Flat GHC that is not equipped with strong moding/typing, but the debugging of KL1 programs turns out to benefit from moding and typing. Furthermore, its compiler KLIC provides a nice platform for our experiments (Chikayama et al., 1994). We have obtained promising results from our experiments with the assistance of other syntactical constraints (Sect. 5).

2. Strong Moding and Typing in Concurrent Logic Programming

We outline the mode system of Moded Flat GHC. The readers are referred to (Ueda and Morita, 1994) and (Ueda, 1996) 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:

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

where *Fun* and *Pred* are the sets of function and predicate symbols, and N_f and N_p are the sets of possible argument positions (numbered from 1) for the symbols f and p, respectively. The disjoint union operator $\sum$ means:

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

The purpose of mode analysis is to find the set of all modes (each of type $P_{Atom} \rightarrow \{in, out\}$) 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. 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 '$\overline{\ \ }$' 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 head h is an atomic formula and guard G and body B are multisets of atomic formulae. Intuitively, each clause is a rewriting rule, where

h is a template matched with a goal to be rewritten; G is a multiset of conditions; and B is a multiset of subgoals that the goal would be rewritten to. The execution of a program starts with a *goal clause* of the form `:-` B, where B is a multiset of atomic formulae representing goals to be reduced concurrently.

Mode constraints imposed by a clause h `:-` G `|` B are summarized in Fig. 1. All rules here embody the assumption that every piece of communication is cooperative. In concurrent logic programming, variables can be considered as communication channels between the head and the body of a clause or between different body goals of a clause. Rule (BV) means exactly one of the occurrences of a variable works as the outlet of information flow and the other occurrences of the variable work as inlets. Variable occurrences in a clause head with the mode value *in*/*out* work conversely as the outlet/inlet of information when viewed from inside the clause. When a variable occurs at most twice, the relation $\mathcal{R}$ in (BV) is reduced to a unary or a binary relation as:

$$\begin{aligned} \mathcal{R}(\{s\}) &\Leftrightarrow s = OUT \\ \mathcal{R}(\{s_1, s_2\}) &\Leftrightarrow s_1 = \overline{s_2} \end{aligned}$$

Rule (HV) comes from the semantics of Flat GHC that multiple occurrences of a variable in a clause head must receive completely identical terms. Rule (GV) means the occurrence of a variable in a clause head is regarded as an inlet if the variable is tested in the guard. Rule (BF) says that a function symbol in a body goal is a source of information to the callee side, while Rule (HF) says that a function symbol in a clause head is a receptor of information from the caller side. 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 (Cho and Ueda, 1996), but in this paper we will not consider general mode polymorphism because whether to have polymorphism is independent of the essence of this work.

As an example, consider a list concatenation (append) program defined as follows:

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

From the clause R_1, we obtain four constraints:

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

(HV) $m/p = \mathit{IN}$, 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', $\forall q \in P_{\mathit{Term}}(m(p'q) = \mathit{in} \Rightarrow m(pq) = \mathit{in})$.

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

(BF) $m(p) = \mathit{in}$, 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

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

Figure 1. Mode constraints imposed by a program clause h `:-` G `|` B or a goal clause `:-` B.

Mode constraint	Rule	Source symbol
$m(\langle \mathsf{a}, 1\rangle) = \mathit{in}$	(HF)	"`[]`"
$m/\langle \texttt{=}_1, 1\rangle = \overline{m/\langle \texttt{=}_1, 2\rangle}$	(BU)	$\texttt{=}_1$
$m/\langle \mathsf{a}, 2\rangle = m/\langle \texttt{=}_1, 1\rangle$	(BV)	`Y`
$m/\langle \mathsf{a}, 3\rangle = m/\langle \texttt{=}_1, 2\rangle$	(BV)	`Z`

From the clause R_2, we obtain eight constraints:

Mode constraint	Rule	Source symbol
$m(\langle \mathsf{a}, 1\rangle) = \mathit{in}$	(HF)	“`.`”
$m/\langle \texttt{=}_2, 1\rangle = \overline{m/\langle \texttt{=}_2, 2\rangle}$	(BU)	$\texttt{=}_2$
$m(\langle \texttt{=}_2, 2\rangle) = \mathit{in}$	(BF)	“`.`”
$m/\langle \mathsf{a}, 1\rangle\langle \texttt{.}, 1\rangle = m/\langle \texttt{=}_2, 2\rangle\langle \texttt{.}, 1\rangle$	(BV)	`A`
$m/\langle \mathsf{a}, 1\rangle\langle \texttt{.}, 2\rangle = m/\langle \mathsf{a}, 1\rangle$	(BV)	`X`
$m/\langle \mathsf{a}, 2\rangle = m/\langle \mathsf{a}, 2\rangle$	(BV)	`Y`
$m/\langle \mathsf{a}, 3\rangle = m/\langle \texttt{=}_2, 1\rangle$	(BV)	`Z0`
$m/\langle \texttt{=}_2, 2\rangle\langle \texttt{.}, 2\rangle = \overline{m/\langle \mathsf{a}, 3\rangle}$	(BV)	`Z`

Here, “a” stands for `append`; “`.`” stands for the list constructor. In total, twelve constraints are obtained from `append`, and they are consistent as a whole. By simplifying the constraints on “$\texttt{=}_k$”, all the constraints can be reduced to six constraints below:

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

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. 2). Mode graphs are a kind of feature graphs (Aït-Kaci and Nasr, 1986) 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 of $m(p)$ ($\downarrow = \mathit{in}$, $\uparrow = \mathit{out}$),
3. each arc is labeled with the pair $\langle \mathit{symbol}, \mathit{arg}\rangle$ of a predicate/function symbol and an argument position, and may have an inversion bubble (denoted “•” in Fig. 2) 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.

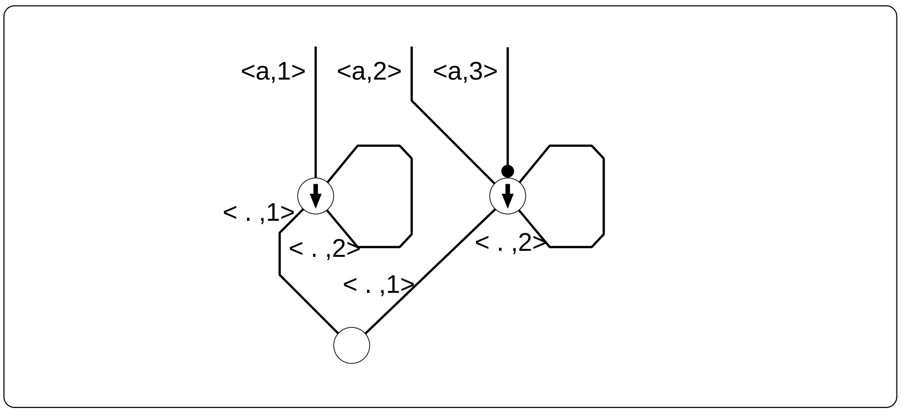

Figure 2. The mode graph of an append program. The mode information of the toplevel predicate and unification goals is omitted.

The cost of mode analysis is almost proportional to the program size for the following reason. Mode analysis proceeds by merging many simple mode graphs representing individual mode constraints. The mode graphs of very large programs are, in general, much wider than that of the append program but are not much deeper, which is to say most nodes can be reachable within several steps from the root. The cost of merging one mode constraint with a mode graph is almost proportional to the depth of the mode graph, but does not depend on the width of the graph (Ueda, 1996). So the total cost is proportional to the number of constraints that in turn is proportional to the program size.

A type system for concurrent logic programming can be introduced by classifying the 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. 3. The choice of a family of sets $F_1, \ldots, F_n$ is arbitrary from the theoretical point of view. This is why moding is more fundamental than typing in concurrent logic programming.

The type system employed by Kima classifies function symbols into six disjoint sets — integers, floating-point numbers, strings, vectors, lists and functor structures, and prohibits any two of them from sharing the same path. Although this is a heuristic classification based on the fact that these different types do not simultaneously appear in the same path in most programs, our experiments prove that it is beneficial both to the power of error detection and to the quality of error correction, as we will see in Sect. 6.

(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$_\tau$) $\tau/\langle \texttt{=}_k, 1\rangle = \tau/\langle \texttt{=}_k, 2\rangle$, for a unification body goal $\texttt{=}_k$.

Figure 3. Type constraints imposed by a program clause h `:-` G `|` B or a goal clause `:-` B.

From our experiences and observations, programmers often make four kinds of simple errors that cannot be trapped as syntactic errors:

1. Typos of variable names — Evident typos are easy to detect even without the declaration of the modes and types of variables. The mode system is sensitive not only to variable occurrences at unexpected positions but also to the loss of variable occurrences. This should be clear by considering how the constraints imposed by rule (BV) (and $\mathcal{R}$) will change when one of two variable occurrences is removed.

2. Confusion of two variables — This is less easy to detect if they are of the same type. Sometimes the error can be corrected using mode info (cf. Example 3 in Appendix); sometimes it results in another meaningful program (cf. Examples 1 in Appendix).

3. Missing body goals — Failure to write necessary body goals (such as those for closing data streams) may cause the loss of variable occurrences, which is likely to be detected by the mode system. However, supplying the missing goal automatically is of course a more difficult task.

4. Missing clauses in predicate definitions — This cannot be detected by using modes and types only, because missing clauses impose no constraints. To detect them would require the analysis of whether the clause guards of a predicate cover all possible cases.

Thus, strong moding can be a useful (if not almighty) tool for the automated debugging of concurrent logic programs to which explicit declarations are usually not provided.

```
$c_{n+1} \leftarrow false$;
$S \leftarrow \{\}$;
**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** $S \leftarrow \{\}$ **fi**
```

Figure 4. Algorithm for computing a minimal inconsistent subset

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 constraints representing the intended specification.

Then, suspicious symbols can be located by computing a minimal inconsistent subset of mode constraints, because the minimal inconsistent subset must include at least one wrong constraint, and each constraint is imposed by certain symbol occurrences in a clause (see the moding rules in Fig. 1). Type constraints can be used in the same way to locate type errors.

A minimal inconsistent subset can be computed efficiently using a simple algorithm shown in Fig. 4 [2]. Let $C = \{c_1, \ldots, c_n\}$ be a multiset of constraints. The algorithm 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.

The readers are referred to (Cho and Ueda, 1996) 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, we can reapply the algorithm to the rest of the constraints.

[2] The algorithm described here is a revised version of the one proposed in (Cho and Ueda, 1996) and takes into account the case when C is consistent.

Our experiment shows that the average size of minimal inconsistent subsets is rather small, and subsets containing more than 10 elements are scarcely found. 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 region of program text. This is due to the redundancy of mode and type constraints. The average size of minimal inconsistent subsets is equal to the number of times of mode analysis that may occur by using the algorithm. So the cost of computing a minimal inconsistent subset is almost proportional to the program size. In reality, the cost is usually much less than the product of the size of a minimal subset and the cost of mode analysis of the whole program owing to an improved algorithm in (Cho and Ueda, 1996).

4. Automated Debugging

Constraints that are considered wrong may 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, that is, by increasing the number of elements of the argument of $\mathcal{R}$ (cf. Rule (BV) of Fig. 1).

When some symbol occurrence has been rewritten to another symbol by mistake, there exists a symbol with less occurrences than intended and a symbol with more occurrences. A minimal inconsistent subset includes either (or both) of them.

Kima focuses on programs with a small number of errors in variables. This focus may sound restrictive, but concurrent logic programs have quite flat syntactic structures (compared with other languages) and instead make heavy use of variables. Our experiences show that a majority of simple program errors arise from the erroneous use of variables, for which the support of static mode and type systems and debugging tools are invaluable.

Other kinds of error correction are not considered by the current version of Kima, but the above technique could be applied also to the correction of the other kinds of symbol occurrences. This is because mode and type constraints are imposed also on constant symbols and function symbols. Mutation from a variable symbol to a constant symbol could be corrected by the same technique. Mutation of constant symbols may be located by type constraints, but its automated correction is difficult. This is because Kima would then have to choose a particular symbol, which is difficult to do based solely on type information. Mutations of function symbols (other than constant symbols) and predicate symbols can also be located but their correction is again difficult, because search space will expand too much in trying to find both appropriate function and predicate symbols and their arguments.

4.1. Basic Algorithm

An algorithm for automated error correction is basically a search procedure whose initial state is the erroneous program, whose operations are the rewriting of the occurrences of variables, and whose final states are well-moded/typed programs.

Given a program L, which is a set $\{l_1, \ldots, l_n\}$ of clauses, let W_L be defined as

$$W_L = \{(i, v) \mid 1 \le i \le n\,,\ v \in V_{l_i}\}$$

where V_{l_i} is the set of variable symbols occurring in the clause l_i.

An element of W_L is called a *located variable*; it is the pair of a clause number and a variable occurring in the clause. Since Kima considers errors in variables, for each minimal inconsistent subset of constraints, we can think of a corresponding set of all located variables that are responsible for the inconsistency. By abuse of language, henceforth we call the latter set a minimal inconsistent subset (of located variables) as well. Let $misv(L)$ represent the minimal inconsistent subset of located variables corresponding to the subset computed by the algorithm in Fig. 4.

The algorithm in Fig. 5 finds a set S of alternative solutions of the program L, where $cls(w)$ $(w \subseteq W_L)$ stands for a set of clause numbers included in w, namely

$$cls(w) = \{i \mid \exists v\ (i, v) \in w\}\ .$$

Given a set $L' = \{l_{k_1}, \ldots, l_{k_m}\} \subseteq L$ of clauses, we can think of an d-mutated set, $\{l'_{k_1}, \ldots, l'_{k_m}\}$, in which d variable occurrences have been mutated from L'. Let $Q^d_{L'}$ represent the set of all d-mutated sets of clauses. The function $mut(q)$ $(q \in Q^d_{L'})$ returns the set of mutated located variables, an element of which is the pair of a clause number $(\le n)$ and a mutated variable symbol (either with more occurrences or with less occurrences). That is, $mut(q)$ represents a subset of W_q. The function $mcs(L)$ stands for mode constraints obtained from the program L; $tcs(L)$ stands for type constraints. The main procedure of the algorithm is iterative-deepening search up to the maximum depth d_{MAX} which is to be given by a user. Note that, instead of iterative-deepening search, depth-first search may also be used because Kima does not discriminate alternatives by the depth where they are found.

```
$w \leftarrow misv(L)$;  $L_1 \leftarrow cls(w)$;
$L_0 \leftarrow L \setminus L_1$;
$S \leftarrow \{\}$;
for $d \leftarrow 1$ to $d_{MAX}$ do
  for each $q \in Q^d_{L_1}$ do
    if $w \cap mut(q) \neq \{\}$ then
      if $mcs(L_0 \cup q)$ is consistent $\wedge$ $tcs(L_0 \cup q)$ is consistent then
        $S \leftarrow S \cup \{q\}$
      fi
    fi
  end for
end for
```

Figure 5. Basic algorithm for automated error correction

Since checking modes and types of a rewritten program requires the cost proportional to the program size (Sect. 2), this algorithm takes time proportional to the program size. However, inconsistency usually occurs within a small region of program text (Sect. 3). A large performance improvement will therefore be achieved by analyzing those constraints imposed by the suspected predicates and predicates closely related to them in the call graph of the program before the whole constraints.

4.2. Grouping Errors

As we mentioned in Sect. 3, multiple minimal inconsistent subsets may independently be found, and some of them may indicate the same clause as the source of errors. The clause may be indicated by subsets of modes, types, or both. Modes and types express different properties of a program and detect different kinds of errors. To use them together makes two improvements; one is that more errors can be detected; the other is that errors can be located more precisely. Kima groups minimal inconsistent subsets indicating the same clause (as in Fig. 6). A group thus formed plays the role of a unit of searching alternatives against errors.

Formally, the grouping of minimal inconsistent subsets means to classify them using the reflexive transitive closure of the following relation, $\rightleftharpoons$, as the equivalence relation:

$$x \rightleftharpoons y \;\Leftrightarrow\; cls(x) \cap cls(y) \neq \{\}$$

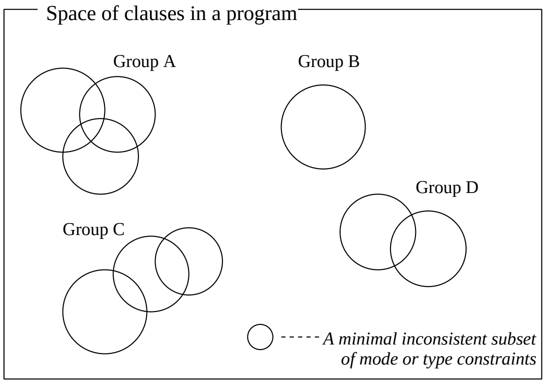
Space of clauses in a program
Group A
Group B
Group C
Group D
A minimal inconsistent subset of mode or type constraints

Figure 6. Grouping minimal inconsistent subsets

The grouping can be implemented in the following way. For each clause l_i mentioned in a minimal inconsistent subset S, we form a pair (i, S). Using all the pairs generated, we can readily make two-way links between subsets indicating the same clause. Next, we classify subsets by tracing the links. The cost of tracing is not a problem because it is unlikely that many subsets intricately overlap with each other.

Depth-δ search of alternatives is carried out independently for each group. Computation time can be reduced by checking whether a certain rewriting can possibly dissolve inconsistencies of all minimal inconsistent subsets in a group before actually computing modes and types, which is called *Quick-check*. This is effective because when some symbol occurrence is rewritten, even if the change is small, mode and type analyses may reanalyze the whole program.

Suppose that n groups of minimal inconsistent subsets have been formed from the program L and that a group G_i contains the subsets $w_{i1}, w_{i2}, \ldots, w_{ig_i}$. Then, a set S_i of alternatives for each group G_i is computed by the algorithm in Fig. 7.

When multiple groups are found, mode and type analyses are performed with the clauses indicated by one of the groups and the consistent part L_0, which is the set of all clauses that are not indicated by any minimal inconsistent subset. Therefore, not all constraints imposed by the whole program text are considered in error correction. Kima employs this strategy so that the search of alternatives for one group may not be influenced by that for another group.

4.3. Complexity

We consider the cost of the basic algorithm in Fig. 5, namely depth-δ search of alternatives from one minimal inconsistent subset. Let u be

```
**for** $i \leftarrow 1$ **to** $n$ **do**
  $L_i \leftarrow \bigcup_{j=1}^{g_i} cls(w_{ij})$;
**end for**;
$L_0 \leftarrow L \setminus \bigcup_{j=1}^{n} L_j$;
**for** $i \leftarrow 1$ **to** $n$ **do**
  $S_i \leftarrow \{\}$;
  **for** $d \leftarrow 1$ **to** $d_{MAX}$ **do**
    **for each** $q \in Q_{L_i}^{d}$ **do**
      **if** $quick\_check(q, i)$ **then**
        **if** $mcs(L_0 \cup q)$ is consistent $\wedge$ $tcs(L_0 \cup q)$ is consistent **then**
          $S_i \leftarrow S_i \cup \{q\}$
        **fi**
      **fi**
    **end for**
  **end for**
**end for**

**function** $quick\_check(q, i)$
  **return** $\bigwedge_{j=1}^{g_i}(w_{ij} \cap mut(q) \neq \{\})$
**end.**
```

Figure 7. Algorithm for automated error correction with grouping

the number of clauses mentioned in the minimal inconsistent subset of located variables, r the number of variable occurrences in one clause, and t the number of different variables in that clause. For simplicity, we assume all clauses have the same r and the same t.

Then, how many rewritten programs will be generated and checked for well-typedness and well-modedness can be described as ${}_{ur}\mathrm{C}_{\delta} \cdot {}_{t+\delta}\mathrm{P}_{\delta}$, which is the number of possible ways of rewriting δ variable occurrences in clauses indicated by a minimal inconsistent subset. ${}_{ur}\mathrm{C}_{\delta}$ is concerned with the choice of variable occurrences to be rewritten, whereas ${}_{t+\delta}\mathrm{P}_{\delta}$ is the maximum number of ways of rewriting δ variable occurrences. The latter is derived from the following observation: because we need to consider rewriting to a fresh variable, there are $t+1$ ways to rewrite the first variable occurrence and the number of different variables may increase by one after the first rewriting.

Suppose the cost of mode and type analyses is $k \cdot n$, where k is a constant, and n is the program size. Then, the cost of the basic algorithm is expressed by the product of the number of programs generated in the search and the cost of mode and type analyses:

$$\begin{aligned} T &\le {}_{ur}\mathrm{C}_{\delta} \cdot {}_{t+\delta}\mathrm{P}_{\delta} \cdot kn \\ &\le \frac{(ur) \cdot (ur-1) \cdot \ldots \cdot (ur-\delta+1)}{\delta\,!} \cdot (t+\delta)^{\delta} \cdot kn \\ &\le k \cdot \frac{(ur(t+\delta))^{\delta}}{\delta\,!} \cdot n \end{aligned}$$

In most cases, u is smaller than 4 from our experiences. The average sizes of r and t are 13 and 6.4, respectively, for the case of Kima system, which is a KL1 program of 4,500 lines long. The branching factor of this search problem is not small, but the number of "plausible" programs is extremely small compared to the number of all programs generated in the search (Sect. 6). We can take advantage of this fact, as well as Quick-check, to save the number of mode and type analyses (Sect. 5.3).

5. Constraints Other Than Modes and Types

5.1. Prioritizing Alternatives

Kima searches alternative solutions using mode and type information, but multiple alternatives are found in many cases. Kima refines the quality of its output by prioritizing alternatives using two *Heuristic Rules*:

Heuristic Rule 1. It is less likely that a variable occurs

1. only once in a clause (singleton occurrence),
2. two or more times in a clause head,
3. three or more times in the head and/or the body of a clause, or
4. two or more times as arguments of the same body goal.

Heuristic Rule 2. It is less likely that a list and its elements are of the same type, that is, it is less likely that a variable occurs both in some path p and in the path of its elements $p\langle .,1\rangle$.

Since variable occurrences falling under Heuristic Rules 1.1, 1.2 and 1.3 impose mode constraints *IN* or *OUT* (Sect. 2) that are stronger than *in* and *out*, we could replace Heuristic Rules 1.1–1.3 by a unified rule on constraint strength: *A solution with weaker mode constraints is more likely to be an intended one.* In general, stronger mode/type constraints make a program less generic, and the execution of the program more likely to end in failure. Therefore it is reasonable to insist that the constraint imposed on a program should be as weak as possible.

Heuristic Rules 1.1 and 1.3 are justified also on the ground that logical variables are used for one-to-one communication more frequently than for one-to-many or one-to-zero communication. A logical variable used for one-to-one communication occurs either exactly twice in a clause body or exactly once in a clause head and once in a clause body. A body goal with arguments as mentioned in Heuristic Rule 1.4 either receives duplicated data from another goal or communicates with itself, which are both unlikely.

The idea behind Heuristic Rule 2 is as follows. Let α be a type variable and $list(\alpha)$ be the list type whose elements are of type α. Then the rule is equivalent to saying that constraint $\alpha = list(\alpha)$ imposes a strong type constraint on α and is therefore unlikely.

Kima prioritizes multiple alternatives by imposing certain penalty points on unlikely symbol occurrences. An alternative with a lower total penalty point has a higher priority.

5.2. Reinforcing Detection Power

The objective of Kima is to debug a program in the absence of explicit declarations of program properties such as modes and types. To enhance the power of the error detection with implicit modes and types, Kima employed the following auxiliary *Detection Rules*:

Detection Rule 1.

1. A variable which occurs in a clause guard must occur also in the head of the clause.
2. The same variable must not occur on both sides of a unification body goal.

Detection Rule 2. The name of a singleton variable must begin with an underscore "`_`".

Both Detection Rules are optional and can be used selectively in Kima.

Violation of Detection Rule 1.1 means the existence of a variable which is never instantiated, while violation of Detection Rule 1.2 means that the unification body goal either fails (e.g. `X=f(X)`) or does nothing meaningful (e.g. `X=X`). Detection Rule 2 is identical to requesting the declaration of variables that impose strong mode constraints. Detection Rule 2 is effective because a logical variable in a correct program is likely to occur twice in a clause (i.e., for one-to-one communication), in which case a variable will turn into a singleton if either occurrence is missing.

The source of an error detected by Detection Rules is a variable symbol in a certain clause, and is found independently of minimal inconsistent subsets of mode and type constraints. Kima uniformly deals with a variable symbol detected by Detection Rules by considering it as a minimal inconsistent subset with one element, and groups it with other subsets.

5.3. Optimizing Search of Alternatives

Kima employs two optimization techniques other than Quick-check. The two techniques are based on prioritizing and Detection Rules stated in Sect. 5.1 and 5.2, respectively, and reduce the number of mode and type analyses in generate-and-test search. The algorithm shown in Fig. 8 computes a set $S_{i,k}$ of alternatives for each group G_i $(1 \leq i \leq n)$ and priority k $(1 \leq k \leq k_{MAX} + 1)$, where $k = 1$ means the highest priority and k_{MAX} is to be given by a user.

Prioritizing with Heuristic Rule 1 is cheaper than mode and type analyses because it involves only suspected clauses. For each i and d, Kima first divides the set $Q^d_{L_i}$ of programs into sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_{h_i}$ by their priorities, where programs in $\mathcal{Q}_1$ have the highest priority. The value of h_i is not known until we actually divide $Q^d_{L_i}$. When we are interested only in high-priority alternatives, this classification saves the number of mode and type analyses. However, Prioritizing with Heuristic Rule 2 needs type analysis, and is performed after the check of well-typedness (i.e., type reconstruction). The function *heuristic_rule2_ok*(L) returns *true* iff the program L contains no variables that are less likely with respect to Heuristic Rule 2.

The function *detection_rules_ok*(L) returns *true* iff the program L observes Detection Rules, and is called before mode and type analyses. The test of Detection Rules is cheaper than mode and type analyses, because the clauses that are not rewritten do not have to be checked again. In contrast, mode and type analyses may need recalculation of the whole program.

For a quicksort program containing two wrong variable occurrences in the same clause (Example 3 in Appendix), the above optimization improved the response time of computing highest-priority alternatives from 25.9 seconds to 10.2 seconds on the KLIC system running on Sun Ultra 30 (248 MHz) + 128 MB of memory.

```
for $i \leftarrow 1$ to $n$ do
  $L_i \leftarrow \bigcup_{j=1}^{g_i} cls(w_{ij})$;
end for;
$L_0 \leftarrow L \setminus \bigcup_{j=1}^{n} L_j$;
for $i \leftarrow 1$ to $n$ do
  for $k \leftarrow 1$ to $k_{MAX} + 1$ do
    $S_{i,k} \leftarrow \{\}$
  end for;
  for $d \leftarrow 1$ to $d_{MAX}$ do
    prioritize $Q^d_{L_i}$ into $\mathcal{Q}_1, \ldots, \mathcal{Q}_{h_i}$ with Heuristic Rule 1;
    $j \leftarrow 1$;  $k \leftarrow 1$;
    while $j \leq h_i \ \wedge \ k \leq k_{MAX}$ do
      for each $q \in \mathcal{Q}_k$ do
        if $quick\_check(q, i)$ then
          if $detection\_rules\_ok(q)$ then
            if $mcs(L_0 \cup q)$ is consistent
               $\wedge$ $tcs(L_0 \cup q)$ is consistent then
              if $heuristic\_rule2\_ok(q)$ then
                $S_{i,k} \leftarrow S_{i,k} \cup \{q\}$
              else
                $S_{i,k+1} \leftarrow S_{i,k+1} \cup \{q\}$
              fi
            fi
          fi
        fi
      end for;
      if $S_{i,k} \neq \{\}$ then
        $k \leftarrow k + 1$
      fi;
      $j \leftarrow j + 1$
    end while
  end for
end for
```

Figure 8. Optimized algorithm for automated error correction

6. Experiments

We discuss the effectiveness of our technique based on experiments. We investigated how many of programs with a couple of errors are detected as erroneous by Kima, how many alternatives it proposes given an erroneous program, and how many "plausible" programs there are in the neighborhood of a correct (original) program.

Sample programs we used are list concatenation (append), the generator of a Fibonacci sequence, and quicksort. They are admittedly simple but the aim of the experiment is to investigate the fundamental power of our technique based on exhaustive experiments. Further, we strongly expect that the total program size does not make much difference in the quality and the performance of automated debugging except, of course, that the cost of analysis is necessarily proportional to the program size and the number of bugs. The main differences between toy programs and non-trivial programs are the number of clauses and the number of (variable) symbol occurrences in a clause. However, the number of clauses does not matter, because the locations of bugs are limited to small regions of program text in advance by finding minimal inconsistent subsets whose size is independent of the program size (Sect. 3). Also, Kima can successfully cope with a program with an error in a large clause; see Example 2 in Appendix.

Experiment 1

First, we systematically generated near-misses (each with one wrong occurrence of a variable) of three programs and examined how many of them could be detected, whether automated correction reported an intended program, and how many alternatives were reported. Table I shows the results[3]. Here, we considered all possible ways of the mutation of a variable occurrence, that is, mutation to a fresh (i.e., singleton) variable as well as mutation to another variable in the same clause, but did not consider mutation to the variable whose name began with "_", which would be very unlikely as human errors. We used only the definitions of predicates in error correction, that is, we did not use the constraints that might be imposed by the caller of these programs. Of course, the caller information, if available, would enhance the quality of correction as well as the redundancy of constraints.

[3] In a similar experiment shown in our previous paper (Ajiro et al., 1998), the numbers are different because (i) errors in the clause guard and those concerning Detection Rule 1 were not counted and (ii) errors detected by types but not detected by modes were not considered by automated debugging.

Table I. Single-error detection and correction

Program	Analysis	Level	Priori-tizing	Total cases	Dete-cted	Proposed Alternatives 1	2	3	4	5	6	≥7
`append`	mode only	0	no	58	36	1	3	8	3	6	5	10
	type only	0	no	58	0	0	0	0	0	0	0	0
	mode & type	0	no	58	36	1	3	8	3	6	5	10
	mode & type	0	yes	58	36	27	9	0	0	0	0	0
	mode & type	1	yes	58	40	29	11	0	0	0	0	0
	mode & type	2	yes	58	58	39	19	0	0	0	0	0
`fibonacci`	mode only	0	no	118	57	10	7	9	6	4	1	20
	type only	0	no	118	47	0	0	4	20	0	18	5
	mode & type	0	no	118	72	18	13	2	15	9	0	15
	mode & type	0	yes	118	72	54	11	1	6	0	0	0
	mode & type	1	yes	118	88	68	12	7	0	1	0	0
	mode & type	2	yes	118	99	71	18	8	0	2	0	0
`quicksort`	mode only	0	no	300	177	34	70	1	12	19	0	41
	type only	0	no	300	106	0	2	12	40	0	32	20
	mode & type	0	no	300	221	49	76	8	59	0	9	20
	mode & type	0	yes	300	221	164	41	16	0	0	0	0
	mode & type	1	yes	300	236	175	61	0	0	0	0	0
	mode & type	2	yes	300	286	199	84	2	1	0	0	0

The column "Level" indicates detection levels. At detection level 0, only mode and/or type information was used; at detection level 1, Detection Rule 1 was used in addition; and at detection level 2, Detection Rules 1 and 2 were used together. The two Detection Rules raised the average detection rate from 69.1% (329/476) to 93.1% (443/476).

A row with "yes" in the column "Prioritizing" shows the number of proposed alternatives with the highest priority. The number of proposed alternatives under prioritizing was usually 1 or quite small. The exceptions were in erroneous `fibonacci` programs. A certain variable occurs four times in the correct `fibonacci` (`N2` in the program in Sect. 7), and there were a few cases where one of its body occurrences was mutated to some fresh variable and their highest-priority corrections were to replace another occurrence of `N2` to that fresh variable.

Table II. Error detection rate for the programs with N mutations

Program	N	Level	Total cases	Detected cases	Detection rate (%)
`append`	2	0	1200	937	78.1
	2	1	1200	1004	83.7
	2	2	1200	1141	95.1
	3	0	16980	14597	86.0
	3	1	16980	15411	90.8
	3	2	16980	16674	98.2
`fibonacci`	2	0	4668	3982	85.3
	2	1	4668	4330	92.8
	2	2	4668	4489	96.2
	3	0	133045	125300	94.2
	3	1	133045	130325	97.9
	3	2	133045	131810	99.1
`quicksort`	2	0	12102	11263	93.1
	2	1	12102	11460	94.7
	2	2	12102	12005	99.2
	3	0	337455	330769	98.0
	3	1	337455	332416	98.5
	3	2	337455	336943	99.8

Experiment 2

Second, we investigated the error detection rate for programs with two or three mutated variable occurrences in the same clause. Errors of this kind are looked on as depth-2 and depth-3 errors in the same group, respectively, and their correct alternatives can be obtained by depth-2 and depth-3 search. Table II shows the results. Note that the mutation of variable occurrences does not always cause errors. For example, certain mutations make a program equivalent to the original as we will see later.

When multiple errors existed in some clause of a program, the error was detected as long as at least one of the errors caused inconsistency. So the detection rate of multiple errors was higher than that of a single error. The detection rate with Detection Rules 1 and 2 was above 95% in every case.

Table III. The number of plausible programs in the programs with N mutations

Program	N	Total cases	Plausible programs	With caller info	Related to guard
`append`	1	58	0	0	0
	2	1200	7	1	0
	3	16980	14	3	0
	4	167842	29	5	0
`fibonacci`	1	118	11	9	4
	2	4668	66	3	3
	3	133045	309	5	5
`quicksort`	1	300	9	4	4
	2	12102	33	17	4
	3	337455	76	48	16

Experiment 3

Third, we explored the number of "plausible" programs in the set of programs with N mutations on variable occurrences in the same clause. By "plausible" we mean programs that have the same or higher priority than the original program. The result is shown in Table III. In this experiment, we investigated not only the possibility for Kima to find an intended program by search, but also what the programs that passed the criteria in our framework looked like.

In order to precisely count the number of all programs with the same (or higher) priority as the original program, here we included the mutation to the variables whose name began with "_", because programs with a singleton variable whose name does not begin with "_" are eliminated immediately by Detection Rules. In the column "Plausible programs", programs that were

1. equivalent up to the renaming of variables and
2. equivalent up to the exchanging of arguments of calls to commutative built-in predicates such as unification

were counted as one program.

From Table III we see that the number of plausible programs did not increase as rapidly as the number of total cases. This can be explained by the fact that the ways of placing variable symbols which make a program well-moded and well-typed are extremely limited compared to arbitrary ways of placing.

Now we focus on the number of proposed alternatives under prioritizing. Suppose, for example, a program contains two errors on variable occurrences and depth-2 search is performed. In this case, programs in the search space have up to four occurrences rewritten from the original, correct program. Of these programs, those with four mutations will be the majority. However, since two of the four mutations have already been done by the given erroneous program, only part of the programs with up to four mutations are generated. The total number of programs generated for inspection is very close to the number of cases with $N = 2$ in Table III.

Caller information (i.e., examples of clauses containing calls to the predicates to be debugged) can reduce the number of plausible programs, because they play the role of mode and type specifications. The column "With caller info" indicates the number of plausible programs in the existence of caller information, where the original, intended programs were not counted. Out of those programs, the number of programs whose guard goals were essentially rewritten is indicated in the column "Related to guard". That is, the rightmost column shows the number of mutated but plausible programs in which either (i) a variable tested in a clause guard was mutated, or (ii) a variable tested in a clause guard was made to occur at a different argument position in the clause head.

More than half of programs that are in the column "With caller info" but not in the column "Related to guard" diverge or cause deadlock. The other programs return unintended output. For example, we have found:

1. a program that merges two input lists by taking their elements alternately in the neighborhood of `append` (cf. Appendix),

2. a program that returns the list of all natural numbers (`[0,1,2, ...]`) in the neighborhood of the generator of a Fibonacci sequence, and

3. a program that sorts list items in descending order in the neighborhood of `quicksort` that sorts list items in ascending order.

Also, there were programs to which we could not find any concise meaning.

7. An Example — Fibonacci sequence

As an example, we consider the generator of a Fibonacci sequence with one error:

R_1 : `fib(Max,_, N2,Ns0):- N2 >Max | Ns0=`$_1$`[].`
R_2 : `fib(Max,N1,N2,Ns0):- N2=<Max |`
    `N1=`$_2$`[N2|Ns1],N3:=N1+N2,fib(Max,N2,N3,Ns1).`
(The body unification in R_2 should be `Ns0=`$_2$`[N2|Ns1]`)

The algorithm shown in Sect. 3 computes three independenet minimal inconsistent subsets; two on modes and one on types. Here, we do not consider Detection Rule 2 (though it can detect the variable `Ns0` in the clause head of R_2 as an error).

Minimal inconsistent subset 1 (on modes):

Mode constraint	Rule	Source symbol
$m(\langle\texttt{=}_1, 2\rangle) = \mathit{in}$	(BF)	"`[]`" in R_1
$m/\langle\texttt{=}_1, 1\rangle = m/\langle\texttt{fib}, 4\rangle$	(BV)	`Ns0` in R_1
$m/\langle\texttt{=}_1, 2\rangle = \overline{m/\langle\texttt{=}_1, 1\rangle}$	(BU)	$\texttt{=}_1$ in R_1
$m(\langle\texttt{fib}, 4\rangle) = \mathit{IN}$	(BV)	`Ns0` in R_2

Minimal inconsistent subset 2 (on modes):

Mode constraint	Rule	Source symbol
$m(\langle\texttt{=}_2, 2\rangle) = \mathit{in}$	(BF)	"`.`" in R_2
$m/\langle\texttt{=}_2, 2\rangle = \overline{m/\langle\texttt{=}_2, 1\rangle}$	(BU)	$\texttt{=}_2$ in R_2
$m(\langle\texttt{=}_2, 1\rangle) = \mathit{IN}$	(BV)	`N1` in R_2

Minimal inconsistent subset 3 (on types):

Type constraint	Rule	Source symbol
$\tau/\langle\texttt{fib}, 2\rangle = \tau/\langle\texttt{:=}, 2\rangle\langle\texttt{+}, 1\rangle$	(HBV$_\tau$)	`N1` in R_2
$\tau(\langle\texttt{=}_2, 2\rangle) =$ list type	(HBF$_\tau$)	"`.`" in R_2
$\tau/\langle\texttt{fib}, 2\rangle = \tau/\langle\texttt{=}_2, 1\rangle$	(HBV$_\tau$)	`N1` in R_2
$\tau/\langle\texttt{=}_2, 2\rangle = \tau/\langle\texttt{=}_2, 1\rangle$	(BU$_\tau$)	$\texttt{=}_2$ in R_2
$\tau(\langle\texttt{:=}, 2\rangle\langle\texttt{+}, 1\rangle) =$ integer type	builtin	`:=` in R_2

These three subsets are classified into the same group because all of the subsets indicate the clause R_2. Suspected variable symbols are extracted as in the table below:

Clause	Variable symbol	Subset number
R_1	`Ns0`	1
R_2	`Ns0`	1
R_2	`N1`	2, 3

When depth-1 search is attempted, Quick-check detects that rewritings which increase or decrease the number of occurrences of `Ns0` in R_1 need not be considered, because such changes may dissolve the subset 1 but neither the subset 2 nor 3. After all, the system finds that the only possible ways to dissolve all inconsistencies are either to replace `Ns0` by `N1` or vice versa in R_2. As the number of occurrences of `Ns0` and `N1` is four in total, only four ways of rewriting each variable occurrence need mode and type analyses. Without Quick-check, a great number of mode and type analyses would have to be done. In this example, Kima finally finds only one alternative, which is the program we have intended.

8. Related Work

Analysis of malfunctioning systems based on their intended logical specification has been studied in the field of artificial intelligence (Reiter, 1987) and known as model-based diagnosis, which has some 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 based on the specification given by the user.

In the field of programming, debugging with partial or abstract specification such as assertions has been studied for many programming languages. For instance, debugging of (concurrent or constraint) logic programs is studied in (Shapiro, 1982; Fromherz, 1993; Puebla et al., 1999).

Type declaration can be thought of as a kind of partial specification. In languages with static typing and automatic type reconstruction, types need not be declared explicitly. This is the approach Kima has employed, but there has been a lot of work on explaining the source of type errors for strongly typed functional languages such as ML (Wand, 1986; Beaven and Stansifer, 1993; Duggan and Bent, 1996; McAdam, 1999; Jung and Michaelson, 2000). When a type error has been found, these systems explain why and how a particular type has been deduced. As far as we can see, they were implemented by extending the unification algorithm for type reconstruction. They recorded which symbol occurrence imposed which constraint in the type deduction process. In contrast, our framework is built outside any underlying framework of constraint solving. It does not incur any overhead for well-moded/typed programs or modify the constraint-solving algorithm. Furthermore, the diagnosis guarantees the minimality of the explanation and often refines it further.

A soft typing approach introduces static type systems to dynamically typed languages such as Lisp and Scheme (Cartwright and Fagan, 1991; Aiken et al., 1994). The MrSpidey system (Flanagan and Felleisen, 1998) has a programming environment that visually presents the explanation of type errors of Lisp programs based on soft typing and set-based analysis (Heintze and Jaffar, 1994). In this approach, debugging is performed interactively because the judgement of whether suspected fragments of a program are really wrong necessarily depends on the programmer. Additional information given by the programmer in the interaction could be thought of as declarations. The choice between static and dynamic approaches is a question of tradeoff between safety and fexibility of program description, but we think static typing approach is suited for large-scale and/or complicated programs in parallel and distributed computing.

Tenma's system automatically corrects Lisp programs under typing (Tenma et al., 1990). When a change is made on a certain software component, the system automatically replaces the components that do not adapt to the change by alternative components. Thus the purpose of the system is very different from Kima. Kima works in the situation where the locations of errors are entirely unknown, and it works at the program symbol (primitive) level rather than the software component level.

Incidentally, we heard from a referee that the old Fortran compilers of the 1970's had been equipped with automated error correction. The input was on punched cards, and since resubmitting a job took a long time, the compilers did go rather far in correcting syntactical errors; in many cases, the correction was correct, and this saved a lot of time.

9. Conclusions and Future Work

We have implemented Kima, a system which automatically corrects near-misses in concurrent logic programs. Kima does not have to be given explicit specifications of program properties.

Experiments showed that, in most cases, one or a few alternatives could be obtained from KL1 programs with a few wrong variable occurrences. This is indebted to theoretically or statistically endorsed heuristics as well as mode and type information. In the set of programs with a few mutated variable occurrences, programs that are both well-moded/typed and with higher priority turn out to be quite rare. Heuristic Rules and Detection Rules do not only improve the power of error detection and the quality of alternatives but also optimizes the search of alternatives.

Specifications or declarations of program properties, if available, will achieve more advanced error correction. Our future plan is to let Kima accept instances of a pair of input and output constraints. We plan to investigate the possibility of automated programming with a relatively small number of examples and strong support of static analysis.

The computation of minimal inconsistent subsets and the following depth-1 search for a program of 100 lines long is completed within several seconds. The example shown in Sect. 7 took about 0.07 seconds on the KLIC system running on Sun Ultra 30 (248 MHz) + 128 MB of memory. The three examples in Appendix took about 0.05 seconds (append), 0.23 seconds (combination), and and 10.2 seconds (quicksort, see Sect. 5.3), respectively.

The function of the error correction of Kima is rather experimental but the function of error detection by computing minimal inconsistent subsets was very useful in developing Kima (KL1 program of 4,500 lines long) itself. Kima is available from
`http://www.ueda.info.waseda.ac.jp/~ajiro/study-e.html` .

Acknowledgement

We appreciate the referees' suggestions to improve our paper. We would also like to thank Mizuhito Ogawa and Khoo Siau Cheng for their information about related work.

References

Aiken, A., E. L. Wimmers, and T. K. Lakshman: 1994, 'Soft Typing with Conditional Types'. In: *Proc. 21st ACM Symp. on Principles of Programming Languages.* pp. 167–173, ACM.

Aït-Kaci, H. and R. Nasr: 1986, 'LOGIN: A Logic Programming Language with Built-In Inheritance'. *J. Logic Programming* **3**(3), 185–215.

Ajiro, Y. and K. Ueda: 2001, 'Kima – an Automated Error Correction System for Concurrent Logic Programs'. *Computer Software* **18**(0), 122–137. In Japanese.

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

Beaven, M. and R. Stansifer: 1993, 'Explaining Type Errors in Polymorphic Languages'. *ACM Letters on Programming Languages and Systems* **2**(1–4), 17–30.

Cartwright, R. and M. Fagan: 1991, 'Soft Typing'. In: *Proc. ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation (PLDI'91).* pp. 268–277, ACM.

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

Cho, K. and K. Ueda: 1996, 'Diagnosing Non-Well-Moded Concurrent Logic Programs'. In: *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96).* pp. 215–229, The MIT Press.

Duggan, D. and F. Bent: 1996, 'Explaining Type Inference'. *Science of Computer Programming* **27**(1), 37–83.

Flanagan, C. and M. Felleisen: 1998, 'A New Way of Debugging Lisp Programs'. In: *40th Anniversary of Lisp (Lisp in the Mainstream).*

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

Heintze, N. and J. Jaffar: 1994, 'Set Constraints and Set-Based Analysis'. In: *Proc. Principles and Practice of Constraint Programming, Second Int. Workshop (PPCP'94).* pp. 281–298, Springer.

Jung, Y. and G. Michaelson: 2000, 'A visualisation of polymorphic type checking'. *J. Functional Programming* **10**(1), 57–75.

McAdam, B. J.: 1999, 'Generalising Techniques for Type Debugging'. In: *First Scottish Functional Programming Workshop.* Also in *Trends in Functional Programming.*

Milner, R.: 1978, 'A Theory of Type Polymorphism in Programming'. *J. of Computer and System Sciences* **17**(3), 348–375.

Puebla, G., F. Bueno, and M. Hermenegildo: 1999, 'Combined Static and Dynamic Assertion-based Debugging of Constraint Logic Programs'. In: *Proc. Logic-based Program Synthesis and Transformation (LOPSTR'99).* pp. 273–292, Springer.

Reiter, R.: 1987, 'A Theory of Diagnosis from First Principles'. *Artificial Intelligence* **32**, 57–95.

Shapiro, E. Y.: 1982, *Algorithmic Program Debugging*, ACM Distinguished Dissertation Series. Cambridge, MA: The MIT Press.

Tenma, T. et al.: 1990, 'A Modification Support System – Automated Correction of Side-Effects Caused by Type Modifications'. In: *Proc. ACM 18th Annual Computer Science Conference (CSC'90).* pp. 154–160, ACM.

Ueda, K.: 1996, 'Experiences with Strong Moding in Concurrent Logic/Constraint Programming'. In: *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*. pp. 134–153, Springer.

Ueda, K.: 1999, 'Concurrent Logic/Constraint Programming: The Next 10 Years'. In: K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren (eds.): *The Logic Programming Paradigm: A 25-Year Perspective*. Springer, pp. 53–71.

Ueda, K. and T. Chikayama: 1990, 'Design of the Kernel Language for the Parallel Inference Machine'. *The Computer Journal* **33**(6), 494–500.

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

Ueda, K. and M. Morita: 1994, 'Moded Flat GHC and Its Message-Oriented Implementation Technique'. *New Generation Computing* **13**(1), 3–43.

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

Appendix

Example 1 – A single error

Consider a list concatenation (append) program with one error:

```
:- module test.
append([],   Y,Z ):- true | Y=Z.
append([A|Y],Y,Z0):- true | Z0=[A|Z],append(X,Y,Z).
```
(The head should have been `append([A|X],Y,Z0)`)

Suppose you want to obtain alternatives with up to priority 100 (i.e., very low priority). Then, command line options should be given as:

```
  % kima +p 100 append.kl1
```

Then, Kima presents six alternatives, all up to priority 3:

```
  ================= Suspected Group 1 =================
         ------------- Priority 1 -------------
append([A|X],Y,Z0):-true|Z0=[A|Z],append(X,Y,Z)
                        in test:append/3, clause No.2
         -----
append([A|Y],X,Z0):-true|Z0=[A|Z],append(X,Y,Z)
                        in test:append/3, clause No.2
         -----
         ------------- Priority 2 -------------
append([A|Y],Y,Z0):-true|Z0=[A|Z],append(Z0,Y,Z)
                        in test:append/3, clause No.2
         -----
         ------------- Priority 3 -------------
append([A|Y],Y,Z0):-true|Z0=[A|Z],append(Y,Y,Z)
                        in test:append/3, clause No.2
         -----
append([A|Y],Y,Z0):-true|Z0=[A|Z],append(A,Y,Z)
                        in test:append/3, clause No.2
         -----
append([A|Y],Y,Z0):-true|Z0=[A|Z],append(Z,Y,Z)
                        in test:append/3, clause No.2
         -----
```

Each alternative is separeted by “-----”. Redundant alternatives with respect to the equivalence mentioned in Sect. 6 are eliminated in reporting alternatives. The two alternatives of priority 1 have the highest priority. The first alternative is the intended one, while the second alternative turns out to be a program that merges two input lists by taking their elements alternately. That is, when invoked as `append([1,2,3],[4,5,6],Out)`, the first alternative returns `[1,2,3,4,5,6]` and the second returns `[1,4,2,5,3,6]`.

Next, let us compute minimal inconsistent subsets (*MIS*s for short) and variable symbol occurrences infringing Detection Rules.

```
% kima +mis append.kl1
< Minimal Inconsistent Subsets of *Mode* constraints >
m/<(test:append)/3,1><cons,2> = IN
       imposed by the rule HV applied to the variable Y
       in test:append/3, clause No.2
m/<(test:append)/3,1> = OUT
       imposed by the rule BV applied to the variable X
       in test:append/3, clause No.2
-----
< Minimal Inconsistent Subsets of *Type* constraints >
 --Constraints are consistent, and there is no MIS--
< Violations of syntactic rules of Detection Level 2 >
singleton(X)
       in test:append/3, clause No.2
-----
```

Minimal inconsistent subsets of mode constraints are obtained first; those of types second. Multiple independent subsets can be computed at once, and each subset is displayed with a separator “-----”. In this example, only one minimal inconsistent subset on modes is found, while type constraints are consistent.

The subset says that variables `X` and `Y` in the second clause of `append` are suspicious. Using this information, Kima searches alternatives by changing the number of occurrences of `X` and/or `Y` in the clause. In addition to the minimal inconsistent subset, the variable `X` is detected as an error by Detection Rule 2. Violations of Detection Rules are reported as follows:

Detection Rule 1.

1. A variable which occurs in a clause guard must occur also in the head of the clause: `var_not_in_the_head(`*the variable*`)`

2. The same variable must not occur on both sides of a unification body goal: `not_pass_occur_check(`*the variable*`)`

Detection Rule 2. The name of a singleton variable must begin with an underscore "_": `singleton(`*the variable*`)`

Example 2 – Multiple, independent errors

The second example is a program `comb(`n`,`r`,Out)` that generates the list of all length-n 0-1 lists in which the 1's occur exactly r times. (Hence the outer list contains ${}_n\mathrm{C}_r$ elements.) For example, `comb(3,2,Out)` returns the list `[[1,1,0],[1,0,1],[0,1,1]]`. Below is the definition of `comb` with two errors:

```
:- module probability.
comb(N,0,C) :- true | init_list(0,N,0,[],C0),C=[C0].
comb(N,N,C) :- true | init_list(0,N,1,[],C0),C=[C0].
comb(N,R,C) :-  N>R |
   N1:=N-1,R1:=R-1,comb(N1,R1,C0),cons_list(1,C0,CC0),
   comb(N1,R,C1),cons_list(0,C1,CC1),append(CC0,CC1,CC).
 (The last invocation should have been append(CC0,CC1,C))
init_list(N,Len,_,L0,L) :- N=:=Len | L0=L.
init_list(N,Len,E,L0,L) :- N < Len |
   L1=[E|L0],N1:=N+1,init_list(N1,Len,E,L1,L).
cons_list(_,[],    L) :- true | L=[].
cons_list(A,[X|Xs],L) :- true |
   L=[[A|X]|L1],cons_list(A,XS,L1).
 (The recursive call should have been cons_list(A,Xs,L1))
append([],   Y,Z ) :- true | Y=Z.
append([A|X],Y,Z0) :- true | Z0=[A|Z],append(X,Y,Z).
```

The default action of Kima is to perform depth-1 search of alternatives with the highest priority using modes, types, and Detection Rules.

```
% kima comb.kl1
 ================= Suspected Group 1 =================
        ------------- Priority 1 -------------
comb(N,R,C):-N>R|
  N1:=N-1,R1:=R-1,comb(N1,R1,C0),cons_list(1,C0,CC0),
  comb(N1,R,C1),cons_list(0,C1,CC1),append(CC0,CC1,C)
                in probability:comb/3, clause No.3
        -----
 ================= Suspected Group 2 =================
        ------------- Priority 1 -------------
cons_list(A,[X|Xs],L):-true|
  L=[[A|X]|L1],cons_list(A,Xs,L1)
                in probability:cons_list/3, clause No.2
        -----
```

There are two Suspected Groups. In this example, Kima first found multiple minimal inconsistent subsets. By analyzing the clauses indicated by the subsets, Kima concluded there were two independent groups. Kima performed depth-1 search of alternatives for each group, and succeeded in finding alternatives that restored the intended program.

Example 3 – Multiple errors in the same group

Last, we consider a quicksort program with two errors in the same clause.

```
:- module main.
quicksort(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),
   Ys2=[X|Ys1],qsort(L,Ys2,Ys3).
 (The body unification goal should have been Ys1=[X|Ys2])
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).
```

Depth-1 search is tried first, but no solution can be found.

```
% kima qsort.kl1
  ================= Suspected Group 1 =================
              Sorry, no alternative is found
```

Now depth-2 search is tried.

```
% kima +d 2 qsort.kl1
  ================= Suspected Group 1 =================
         ------------- Priority 1 -------------
qsort([X|Xs],Ys0,Ys3):-true|
  part(X,Xs,S,L),qsort(S,Ys0,Ys1),
  Ys1=[X|Ys2],qsort(L,Ys2,Ys3)
                        in main:qsort/3, clause No.2
         -----
```

Only one alternative is found, and this is the intended one. In depth-2 search, depth-1 search is also executed, and all the alternatives found by depth-1 and depth-2 searches are prioritized together. In general, depth-N search includes depth-k search for all $k \leq N$.

Address for Offprints:
Waseda University
Ueda Laboratory (61-410)
3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan