Automated Programming Framework Using Constraint-Based Static Analysis

by

Yasuhiro Ajiro

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Information and Computer Science

Graduate School of Science and Engineering
Waseda University

March 2002

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

Abstract

We propose an automated programming framework using a constraint-based, static type system. Our framework infers a correct form of a program from an almost correct but incomplete version of it. This is done with the guideline of the consistency of several program properties imposed by the type system. Furthermore, thanks to the simplicity of the type system, the framework is compatible with other automation techniques such as programming by examples, which can also be used for the specification of types.

There are at least two possibilities of applying this framework. One is automated error correction in the absence of explicit declarations of types, and this is the main focus of the thesis. There is an intimate relationship between automated debugging and automated programming in the sense that erroneous programs are the most straightforward specification of programs for experienced programmers to provide. The other is the enhancement of automated debugging (i.e. automated programming) with a few instances of input and output constraints of programs, which are admittedly the most natural and simplest specification of the programs in addition to the erroneous programs.

As a practical application of our framework, 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 of 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 error 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 thesis presents the basic algorithm and various optimization techniques implemented in Kima, and then discusses both its effectiveness and progress to automated programming based on qualitative and quantitative experiments.

Acknowledgments

This thesis could not have been written without the intensive support of my supervisor, Kazunori Ueda. My technical manner of research owes to him. If people find well manners in my writing or research talk, Kazunori Ueda has a key role in its achievement. Also, the principal basis of this work, a concurrent logic language with a static mode system (Chapter 2) is attributed to him. The error locating technique (Chapter 3), which is another (important) basis of the work, is attributed to Kenta Cho.

I would like to thank Mizuhito Ogawa, Khoo Siau Cheng, and Zhenjiang Hu for their information about related work. I would also like to thank Norio Kato for his comments on a fault in an algorithm for computing minimal inconsistent subsets. I am also thankful to thesis committee members, Setsuo Ohsuga, Yoshihiko Futamura, Katsuhiko Kakehi, Yoshiaki Fukazawa, et al. for their time and patience. Also, many (anonymous) referees contributed to the improvement of this thesis via conferences and paper submissions.

The author is indebted to student administrators for their continual maintenance of the computing environment of our laboratory. Much of the administrative work is carried out by volunteer service, which is really invaluable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A theoretical framework is indispensable for complicated software to be developed easily and correctly. The chaotic augmentation of software size is due to the improvement of CPU performance and to the distribution of software over parallel and network computers, while the improvement made most CPU power of even personal computers lie idle. The surplus computing power should contribute to the safety and productivity of software as well as produced a predictive interface studied in the field of user interface. Kima system we have implemented as an application of our framework is the first step toward a predictive interface (that is to complement the details) of programs at a program text level.

We propose an automated programming framework that predicts the *correct* forms of programs by using "aggressively" a static program analysis technique with the high computing power. In contrast, so far type system and abstract interpretation on behalf of static analysis have been used, as it were, "defensively" for detecting errors and for the reasoning of program properties assuming that the programs are correct. Kima is an automated error correction system for concurrent logic programs based on static, constraint-based type and mode systems in the absence of programmers' declarations. So, we mean type and mode correctness by correct here.

Automated programming is a technique that infers the precise form of a program from ambiguous specifications such as instances of a pair of input and output of the program, a sequence of trace information, program behaviors written in a natural language, a program schema that is the skelton of a program, etc. Automated programming was actively studied in the 1970's to the early 1980's, but now is not because of the restriction of

its applicability. The target user of the classical automated programming seemed to be novice programmers, whereas the most accesssible representation for experienced programmers is an executable program written in programming languages. Automated debugging is hence a kind of automated programming in the sense that erroneous programs are the straightforward specifications of programs for experienced programmers to give.

Types can be thought of as values on the simplest abstract domain built in a laguage specification, and type analysis is the simplest abstract interpretation. In thise sense, type system is eligible for the foundation of analytic automated programming. In addition, thanks to the simplicity and the close relationship with a language specification, types are compatible with other techniques of automated programming such as programming by examples, which can immediately be used for type specifications. Instances of a pair of input and output constraints allow programs to be more ambiguous as well as enhance the quality of automated debugging by reducing the cost of searching intended programs. For example, few instances and program schema in which variable occurrences are missing might be sufficient for programming.

1.1 Concurrent Logic Programming

The mechanism of error correction in Kima is based on the mode and type systems of Moded Flat GHC [35, 36]. Moded Flat GHC is a concurrent logic (and consequently, a concurrent constraint) language with a constraint-based mode system. Concurrent logic languages [32] 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 [15], 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 [36]. 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.

The constraint satisfaction problem can be solved mostly (though not entirely) as a unification problem over feature graphs (feature structures with cycles) and can be solved in almost linear time with respect to the size of the program [2]. As we will see later, types in Moded Flat GHC also can be reconstructed using a similar (and simpler) technique.

1.2 Debugging by Constraint-Based Mode and Type System

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

[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.

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 [8]. A minimal inconsistent subset can be thought of as a minimal "explanation" of the reason of inconsistency. Furthermore, since each constraint keeps track of the symbol occurrence(s) in the program that imposed the constraint, a minimal subset tells possible sources (i.e., symbol occurrences) of mode errors.

The inconsistency comes from 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.

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

1.3 Objectives

The main objective of this work is to develop an automated error correction system under strong moding and typing by extending the technique of locating errors. We suppose the situation where neither declaration nor specification of programs is available in order to investigate the potential of

strong moding and typing for automated programming. Another objective is, as the extension of the automated error correction, the consideration of a fundamental framework for automated programming. We plan to use a few instances of a pair of input and output constraints of programs as the simplest program specification in addition to mode and type analyses, which are the simplest technique of program analysis.

Although the framework is quite general, whether it is practical or not may depend on the choice of a language. We targets the error correction of KL1 [34] programs 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 [6].

1.4 Contributions

We have implemented the Kima system, which automatically corrects wrong occurrences of logical variables in KL1 programs under strong moding and typing in the absence of the declarations of programs. The function of the Kima sounds quite restrictive, but is justified in the sense that most simple errors are caused by the wrong usage of variable symbols, which are used heavily in (concurrent) logic programs. Even if errors go out of this range of error correction, most errors can be located by minimal inconsistent subsets of mode/type constraints as mode/type errors, which are humanly hard to locate their sources.

Using the information of possible locations of bugs, automated error 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. In this framework, modes are more fundamental than types, but types are concerned with aspects of program properties that are different from modes, and can be used together with modes to reduce the search space, and improve the quality of error correction.

In terms of the semantics of the mode system, we have found there are syntactical constraints which *plausible* programs should observe. The syntactical constraints we have presented as heuristic rules turned out to

be quite effective not only for multiple alternatives proposed by Kima to be reduced but also for the optimization of searching alternatives to errors. Other than heuristic rules, Kima uses two optimization techniques based on grouping minimal inconsistent subsets and the locality of mode/type constraints. These optimizations enables the correction of two or three wrong occurrences of variables in non-trivial programs.

There are two kinds of errors, independent errors whose sources are apart from each other and interdependent errors whose sources impose mode/type constraints related closely. First, independent errors do not cause serious problems of either the quality or the efficiency of error correction. Thanks both to the incremental nature of constraint-based analysis and to the redundancy of mode/type constraints explained above, even large programs can be analyzed separately, and independent errors can be coped with independently of the number of errors. It is a significant feature of our framework that the framework works on a fragment of a program such as a set of predicate definitions in a particular module.

Second, the efficiency of the correction of interdependent errors deeply depends on the depth (i.e., the number of errors in this case) of the search performed for the error correction. However, interdependent errors are of benfit to error detection, because more errors in a neighboring position can more possibly cause mode/type errors. An experiment showed that the error detection rates of a single error and two interdependent errors were 90% and 95%, respectively.

1.5 Overview of the Thesis

The rest part of the thesis is organized as follows. Chapter 2 outlines strong moding and typing in Moded Flat GHC. Mode/type analysis of the Moded Flat GHC is a constraint satisfaction problem of mode/type constraints imposed by program text. Chapter 3 shows some algorithms for computing minimal inconsistent subsets from constraints inconsistent as a whole. Since each constraint is imposed by symbol occurrences in a program clause, the minimal subsets locate the possible sources of the inconsistency resulting from errors. Chapter 4 presents the technique of correcting near-misses located by the minimal inconsistent subsets. Alternatives to errors are computed basically by generate-and-test search. We also show the syntactical constraints prescribing the plausibility of programs, which can be used not

only to enhance the quality of alternatives but also to optimize the search. Additionally, two other optimization techniqes are presented based on the grouping of minimal inconsistent subsets and the locality of mode and type constraints, respectively. Chapter 5 shows some experiments with which we discuss the effectiveness of the above techniques. Out of examples in the experiments, we pick up three sample programs including errors to illustrate the actual behavior of automated error correction. We conclude in Chapter 6, where we also refer to related work. The potentiality of our framework is then considered toward the advanced automation of programming processes as future work.

Chapter 2

Moded Flat GHC and Strong Moding and Typing

This chapter outlines the mode and type system of Moded Flat GHC. We refer to the analysis cost and polymorphism after the definition of the mode and type system. The readers are referred to [29, 30, 36, 31] for details.

2.1 Moded Flat GHC

Moded Flat GHC is a concurrent logic language where a mode system has been introduced into Flat GHC, a subset of Guarded Horn Clauses (GHC). GHC has a lot of similarities to Prolog, a logic language, but they are essentially different languages. GHC is a "reactive" language while Prolog is a transformational language. In GHC programs, a rewrite rule says that a process is reduced into (i.e., defined by) sub-processes. Reduction here can be thought of both as rewriting in a logic programming sense and as dividing a process in a parallel/concurrent programming sense. In concurrent logic programming, logical variables play important roles in communication and synchronization; writing and reading the values of variables are *send* and *receive* of information, respectively; checking the value of variables (in a clause guard) is *wait* for the value to be instantiated.

2.1.1 Syntax

GHC

GHC borrows from logic and logic programming many notions, — variables, function(symbol)s, constants (regarded as 0-ary functions), predicate(sym-

bol)s, terms, atom(ic formula)s, substitutions, renaming, unification.

A Flat GHC program is a set of *guarded clauses*. A guarded clause is of the form

$$h \texttt{ :- } G \texttt{ | } B,$$

where h is an atom, and G and B are multisets of atoms. h is called the *head* of the clause; atoms in G are called *guard goals*; and atoms in B are called *body goals*. The part before the *commitment operator* "|" is called the *guard*, and the part after "|" is called the *body*.

A clause with an empty body is called a *unit clause*. The set of all clauses in a program whose heads have the predicate symbol p is called the *procedure* for p. A goal with the predicate symbol p is said to *call* p.

Informally, each guarded clause is a conditional rewrite rule of goals, where

- h is the template that should match a goal (say g) to be rewritten,
- G is the auxiliary condition for the rewriting (G must be executed without instantiating g), and
- B is the multiset of (sub)goals to replace g.

That a program is a set means that the duplication (up to renaming of variables) of guarded clauses is insignificant, not to mention their ordering. On the other hand, G and B are multisets because two syntactically identical goals may behave differently due to indeterminacy.

To run a program, we use a goal clause of the form

$$\texttt{:- } B,$$

which specifies the initial multiset of body goals.

A goal is either a unification goal of the form $t_1 \texttt{=} t_2$ or a non-unification goal. A unification goal, whose behavior is predefined in the language, may generate a substitution and constrain the possible values of variables. A non-unification goal is rewritten to other goals using guarded clauses, possibly after observing a substitution. The guard of a guarded clause specifies what substitution should be observed before rewriting, and provides the language with a synchronization mechanism.

Flat GHC

The above definition of GHC allows any atom to occur as a guard goal. However, this proved to be unnecessarily expressive as a concurrent language. We first define a class of predicates, called *test predicates*, that are appropriate for the purpose. A predicate p is called a *test predicate* if the procedure for p is defined by a set of unit clauses. Calls to a test predicate have a property that they do not generate observable substitutions; the only thing that matters is whether they succeed or not.

A Flat GHC program is a set of *flat guarded clauses*, clauses in which guard goals are restricted to unification goals and calls to test predicates. Note that an empty set of atoms is described as `true`.

2.1.2 Operational Semantics

Now we formalize the operational semantics of Flat GHC. We follow the structural approach of Plotkin [20], which is now a standard way of describing operational semantics formally. The structural operational semantics of full GHC is found in [24].

Let B be a multiset of goals, and C a multiset of equations that represents a (binding) environment of B. Let $\mathcal{V}_F$ denote the set of all variables occurring in a syntactic entity F. The current *configuration* is a triple, denoted $\langle B, C\rangle : V$, such that $\mathcal{V}_B \cup \mathcal{V}_C \subseteq V$. It records the goals to be reduced and the current environment, as well as the variables already in use for the current computation. A computation under a program P starts with the initial configuration $\langle B_0, \emptyset\rangle : \mathcal{V}_{B_0}$, where B_0 is the body of the given goal clause.

What we are going to define is a transition relation $c_1 \longrightarrow c_2$, which reads "the configuration c_1 can be reduced to the configuration c_2." When we need to explicitly mention the program P being used, we use the form $P \vdash c_1 \longrightarrow c_2$, which reads "under the program P, c_1 can be reduced to c_2." By $\stackrel{*}{\longrightarrow}$ we denote the reflexive, transitive closure of $\longrightarrow$. The natural deduction form

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

says that if the transition t_1 can happen under $\mathcal{P}_1$ and the condition Cond holds, the transition t_2 can happen under $\mathcal{P}_2$. The numerator and the condition are omitted if they are empty.

- $\forall .(\neg(f(\mathtt{X}_1,\ldots,\mathtt{X}_m) = g(\mathtt{Y}_1,\ldots,\mathtt{Y}_n)))$, for all pairs f, g of distinct functions (including constants).
- $\forall .(\neg(t = \mathtt{X}))$, for each term t other than and containing $\mathtt{X}$.
- $\forall .(\mathtt{X} = \mathtt{X})$.
- $\forall .(f(\mathtt{X}_1,\ldots,\mathtt{X}_m) = f(\mathtt{Y}_1,\ldots,\mathtt{Y}_m) \Rightarrow \bigwedge_{i=1}^{m}(\mathtt{X}_i = \mathtt{Y}_i))$, for each function f.
- $\forall .(\bigwedge_{i=1}^{m}(\mathtt{X}_i = \mathtt{Y}_i) \Rightarrow f(\mathtt{X}_1,\ldots,\mathtt{X}_m) = f(\mathtt{Y}_1,\ldots,\mathtt{Y}_m))$, for each function f.
- $\forall .(\mathtt{X} = \mathtt{Y} \Rightarrow \mathtt{Y} = \mathtt{X})$
- $\forall .(\mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z} \Rightarrow \mathtt{X} = \mathtt{Z})$

Figure 2.1: Clark's equality theory $\mathcal{E}$, in clausal form

We have three rules. In the following rules, $F \models G$ means that G is a logical consequence of F. $\forall \mathcal{V}_F \,.\, F$ and $\exists \mathcal{V}_F \,.\, F$ are abbreviated to $\forall \,.\, F$ and $\exists \,.\, F$, respectively. Also, following [24], we denote $\exists(\mathcal{V}_F \setminus V) \,.\, F$ by $\delta V \,.\, F$, where V is a finite set of variables. We assume that there is an injection, denoted '$\overline{\cdot}$', from the set of predicates to the set of functions, which is naturally extended to an injection from the set of atoms to the set of terms. $\mathcal{E}$ denotes Clark's equality theory (Figure 2.1).

$$\frac{\mathcal{P} \vdash \langle B_1, C_1\rangle : V \longrightarrow \langle B_1', C_1'\rangle : V'}{\mathcal{P} \vdash \langle B_1 \cup B_2, C_1\rangle : V \longrightarrow \langle B_1' \cup B_2, C_1'\rangle : V'} \tag{i}$$

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

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

Rule (i) expresses concurrent reduction of a multiset of goals. Rule (ii) says that a goal b can be reduced using a guarded clause "$h_i \,\mathtt{:-}\, G_i \mid B_i$" if the head unification $\overline{b} = \overline{h_i}$ and the guard goals G_i can be reduced out without affecting the variables in b. This means that the head unification is restricted

to matching effectively. The condition $\mathcal{V}_{(h_i,G_i,B_i)} \cap V = \emptyset$ guarantees that the guarded clause has been renamed using fresh variables. Rule (iii) says that a unification goal simply publishes (or posts) a constraint to the current environment.

2.2 Mode System

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:

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

Formally, a mode is defined as:

$$P_{Atom} \to \{in, out\} \, ,$$

and the purpose of mode analysis is to find the set of all modes 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

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

We also define IN and OUT as submodes returning *in* and *out*, respectively, for any path:

$$\forall q \in P_{Term}\,(IN(q) = in)\,,$$
$$\forall q \in P_{Term}\,(OUT(q) = out)\,.$$

An overline '$-$' inverts the polarity of a mode, a submode, or a mode value as:

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

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 Figure 2.2.

2.2.1 Rationale of Moding Rules

All rules in Figure 2.2 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. The relation $\mathcal{R}$ in (BV) satisfies the relations below:

$$\mathcal{R}(\{s\}) \;\Leftrightarrow\; s = OUT \tag{2.1}$$
$$\mathcal{R}(\{s_1, s_2\}) \;\Leftrightarrow\; s_1 = \overline{s_2} \tag{2.2}$$

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

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

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

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

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

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

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

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

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

Figure 2.2: Mode constraints imposed by a program clause h :- G | B or a goal clause :- B.

$$\mathcal{R}(\{IN\} \cup S) \Leftrightarrow \mathcal{R}(S) \quad (2.3)$$

$$\mathcal{R}(\{OUT\} \cup S) \Leftrightarrow \forall s' \in S\ (s' = IN) \quad (2.4)$$

$$\mathcal{R}(\{s, s\} \cup S) \Leftrightarrow s = IN \wedge \mathcal{R}(S) \quad (2.5)$$

$$\mathcal{R}(\{\overline{s}, s\} \cup S) \Leftrightarrow \forall s' \in S\ (s' = IN) \quad (2.6)$$

$$\mathcal{R}(\{\overline{s}\} \cup S_1) \wedge \mathcal{R}(\{s\} \cup S_2) \Rightarrow \mathcal{R}(S_1 \cup S_2) \quad (2.7)$$

$$\mathcal{R}(\bigcup_{1 \le i \le n} \{s_i\}) \Rightarrow \mathcal{R}(\bigcup_{1 \le i \le n} \{s_i/q\}), \quad q \in P_{Term} \quad (2.8)$$

Since most variables occur at most twice in a clause, the relation $\mathcal{R}$ is reduced to the unary relation (2.1) or the binary relation (2.2). When some constraints remain non-binary after solving all unary or binary constraints, Kima assumes that nonlinear variables (i.e., variables occurring more than twice in a clause) involved are used for one-way multicasting rather than bidirectional communication. Thus, if a nonlinear variable occurs at p and $m(p)$ is known to be *in* or *out*, Kima imposes a stronger constraint $m/p = IN$ or $m/p = OUT$, respectively. This means that a mode graph computed by Kima is not always 'principal', but the strengthening of constraints reduces most non-binary constraints to unary ones. Our observation is that virtually all nonlinear variables have been used for one-way multicasting and the strengthening causes no problem in practice [33].

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.

2.2.2 Mode Polymorphism

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

Suppose polymorphic predicates should be declared, mode polymorphism can be incorporated quite easily. For polymorphic predicates, their principal mode (i.e., mode graphs) are computed first. To allow different instantiations of a principal mode, a *copy* of the mode graph representing the principal mode is created for each call to a polymorphic predicate, which is then be merged into the mode graph of the whole program.

This technique has been implemented in Kima as for built-in predicates including the test predicates of guard goals. All built-in predicates are distinguished by numbering their occurrences similarly to the case of unification goals. Since the principal modes of guard goals have been computed in advance, the implication in Rule (GV) can readily be reduced to the unary relation. Although the general polymorphism for user-defined polymorphic predicates is yet to be implemented, whether to have polymorphism is independent of the essence of this work.

In the absence of the declaration of polymorphic predicates, the above treatment of polymorphism requires that the mode graphs of polymorphic predicates be obtained before analyzing the rest of the program that uses the polymorphic predicates. So Moded Flat GHC programs should be *stratified* with respect to caller-callee relationship between predicates so that polymorphic predicates can be detected in advance [8].

However, from experiences with the early version of Kima, if we deal with all predicates called from multiple predicates as polymorphic, the power of error detection by mode analysis proved to be rather corrupted. Since the number of predicates used polymorphically is quite limited, to require the declaration of polymorphic predicates might be reasonable also from the viewpoint of the cost of the stratification.

2.2.3 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 Clause R_1, we obtain four constraints:

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

From Clause R_2, we obtain eight constraints:

Mode constraint	Rule	Source symbol
$m(\langle \mathtt{a}, 1\rangle) = \overline{in}$	(HF)	"."
$m/\langle \mathtt{=}_2, 1\rangle = \overline{m/\langle \mathtt{=}_2, 2\rangle}$	(BU)	$\mathtt{=}_2$
$m(\langle \mathtt{=}_2, 2\rangle) = in$	(BF)	"."
$m/\langle \mathtt{a}, 1\rangle\langle \mathtt{.}, 1\rangle = m/\langle \mathtt{=}_2, 2\rangle\langle \mathtt{.}, 1\rangle$	(BV)	A
$m/\langle \mathtt{a}, 1\rangle\langle \mathtt{.}, 2\rangle = m/\langle \mathtt{a}, 1\rangle$	(BV)	X
$m/\langle \mathtt{a}, 2\rangle = m/\langle \mathtt{a}, 2\rangle$	(BV)	Y
$m/\langle \mathtt{a}, 3\rangle = m/\langle \mathtt{=}_2, 1\rangle$	(BV)	Z0
$m/\langle \mathtt{=}_2, 2\rangle\langle \mathtt{.}, 2\rangle = \overline{m/\langle \mathtt{a}, 3\rangle}$	(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 "$\mathtt{=}_k$", all the constraints can be reduced to six constraints below:

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

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 (Figure 2.3). Mode graphs are a kind of feature graphs [2] 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 $m(p)$ ($\downarrow = in$, $\uparrow = out$),
3. each arc is labeled with the pair $\langle symbol, arg\rangle$ of a predicate/function symbol and an argument position, and may have an inversion bubble (denoted "•" in Figure 2.3) that inverts the interpretation of the mode values of the paths beyond that arc, and

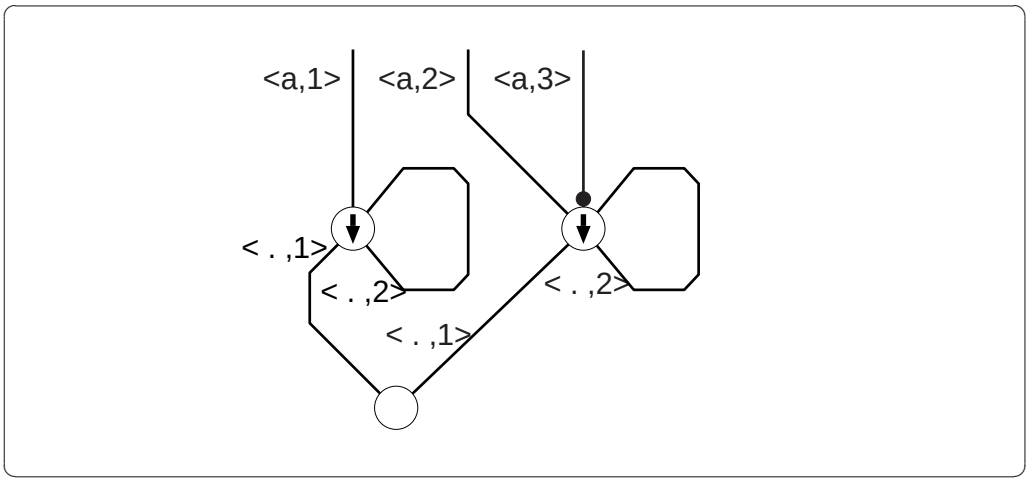

Figure 2.3: The mode graph of an append program. The mode information of the toplevel predicate and unification goals is omitted.

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.

2.2.4 Cost

The cost of mode analysis is almost proportional to the program size. To be precise, the time complexity is $O(nd \cdot \alpha(n))$, where α is the inverse of the Ackermann function; n is the size of the program; and d is the size of the subgraph of the entire mode graph rooted at each predicate argument [36].

Mode analysis proceeds by merging many simple mode graphs representing individual mode constraints. From the observations of various GHC/KL1 programs, the mode graphs of very large programs are, in general, much wider than that of the append program but are not much deeper. Although larger programs have larger mode graphs because they use more predicate symbols, the value of d does not become so large (say several tens of nodes) even for programs using quite complicated communication protocols. Thus we expect that the mode graphs of very large programs are, in general, wide and shallow, which is to say most nodes can be reachable within several steps from the root.

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

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

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

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

(BU_τ) $\tau/\langle =_k, 1\rangle = \tau/\langle =_k, 2\rangle$, for a unification body goal $=_k$.

Figure 2.4: Type constraints imposed by a program clause h :- G | B or a goal clause :- B.

width of the graph [31]. So the total cost is proportional to the number of constraints that in turn is proportional to the program size.

2.3 Type System

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 Figure 2.4. 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 as shown in Table 2.1 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 Section 5.1.1.

2.3.1 An Example

Consider again the `append` program:

Table 2.1: Classification of function symbols in Kima. The *Atom* in F_6 does not include the empty list "`[]`".

Set	Function symbols	Wrapped term form in KLIC
F_1	integers	`integer(`*Int*`)`
F_2	floating-point numbers	`floating_point(`*Float*`)`
F_3	strings	`string(`*Str*`)`
F_4	vectors	`vector({`*Elem, ...*`})`
F_5	lists	`list([`*Car*`|`*Cdr*`])` or `atom([])`
F_6	functor structures	`functor(`*Functor*`(`*Arg, ...*`))` or `atom(`*Atom*`)`

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

From Clause R_1, we obtain four type constraints:

Type constraint	Rule	Source symbol
$\tau(\langle \mathtt{a}, 1\rangle) =$ list type	(HBF_τ)	"`[]`"
$\tau/\langle \mathtt{=}_1, 1\rangle = \tau/\langle \mathtt{=}_1, 2\rangle$	(BU_τ)	$\mathtt{=}_1$
$\tau/\langle \mathtt{a}, 2\rangle = \tau/\langle \mathtt{=}_1, 1\rangle$	(HBV_τ)	`Y`
$\tau/\langle \mathtt{a}, 3\rangle = \tau/\langle \mathtt{=}_1, 2\rangle$	(HBV_τ)	`Z`

From Clause R_2, we obtain eight type constraints:

Type constraint	Rule	Source symbol
$\tau(\langle \mathtt{a}, 1\rangle) =$ list type	(HBF_τ)	"`.`"
$\tau/\langle \mathtt{=}_2, 1\rangle = \tau/\langle \mathtt{=}_2, 2\rangle$	(BU_τ)	$\mathtt{=}_2$
$\tau(\langle \mathtt{=}_2, 2\rangle) =$ list type	(HBF_τ)	"`.`"
$\tau/\langle \mathtt{a}, 1\rangle\langle ., 1\rangle = \tau/\langle \mathtt{=}_2, 2\rangle\langle ., 1\rangle$	(HBV_τ)	`A`
$\tau/\langle \mathtt{a}, 1\rangle\langle ., 2\rangle = \tau/\langle \mathtt{a}, 1\rangle$	(HBV_τ)	`X`
$\tau/\langle \mathtt{a}, 2\rangle = \tau/\langle \mathtt{a}, 2\rangle$	(HBV_τ)	`Y`
$\tau/\langle \mathtt{a}, 3\rangle = \tau/\langle \mathtt{=}_2, 1\rangle$	(HBV_τ)	`Z0`
$\tau/\langle \mathtt{=}_2, 2\rangle\langle ., 2\rangle = \tau/\langle \mathtt{a}, 3\rangle$	(HBV_τ)	`Z`

In total, twelve type constraints are obtained from `append`, and they are consistent as a whole. By simplifying the constraints on "$\mathtt{=}_k$", all the constraints can be reduced to six constraints below:

$$\begin{aligned}
\tau(\langle \mathtt{a}, 1\rangle) &= \text{list type} \\
\tau/\langle \mathtt{a}, 1\rangle\langle ., 2\rangle &= \tau/\langle \mathtt{a}, 1\rangle \\
\tau(\langle \mathtt{a}, 2\rangle) &= \text{list type} \\
\tau/\langle \mathtt{a}, 2\rangle\langle ., 2\rangle &= \tau/\langle \mathtt{a}, 2\rangle \\
\tau/\langle \mathtt{a}, 3\rangle &= \tau/\langle \mathtt{a}, 2\rangle \\
\tau/\langle \mathtt{a}, 3\rangle\langle ., 1\rangle &= \tau/\langle \mathtt{a}, 1\rangle\langle ., 1\rangle
\end{aligned}$$

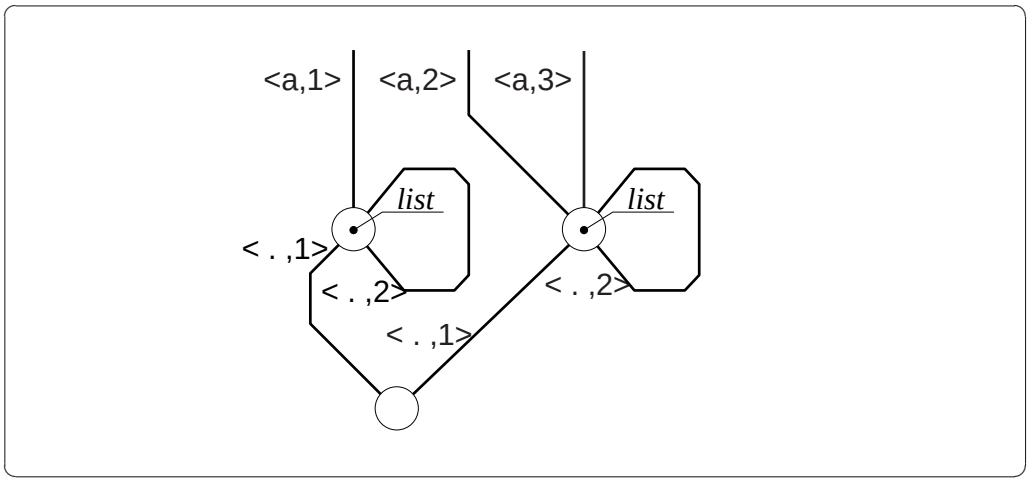

Figure 2.5: The type graph of an append program. The type information of the toplevel predicate and unification goals is omitted.

Similarly, we could regard these constraints themselves as representing the principal type of the program, and the principal type can be represented more explicitly in terms of a type graph (Figure 2.5), in which

1. a path represents a member of P_{Atom},
2. the node corresponding to a path p represents the value of $\tau(p)$,
3. each arc is labeled with the pair $\langle symbol, arg \rangle$ of a predicate/function symbol and an argument position, and
4. a binary constraint of the form $\tau/p_1 = \tau/p_2$ is represented by letting p_1 and p_2 lead to the same node.

As shown in Figures 2.3 and 2.5, the mode and type graphs of a program are often very similar. However, modes and types express different properties of a program and detect different kinds of errors (Section 4.3).

Chapter 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 as in Figure 3.1, 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 Figure 2.2). Type constraints can be used in the same way to locate type errors.

This chapter shows several algorithms for computing minimal inconsistent subsets. The readers are referred to [8] for a proof of the minimality of inconsistent subsets obtained by the algorithms below, as well as extensions for mode polymorphism, and other details.

3.1 Kinds of Simple Errors

From the 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

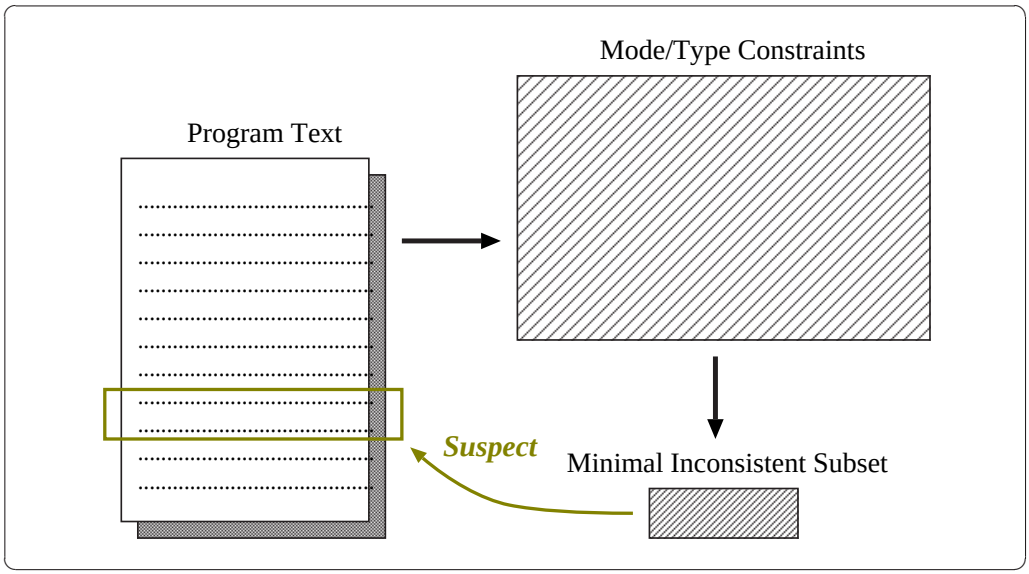

Figure 3.1: Minimal inconsistent subset locates the possible source of an error

$\mathcal{R}$) 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 information (cf. Section 5.2.3); sometimes it results in another meaningful program (cf. Section 5.2.1).

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.

$c_{n+1} \leftarrow \mathit{false}$;
$S \leftarrow \{\}$;
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 3.2: Basic Algorithm for computing a minimal inconsistent subset

3.2 Locating Bugs by Computing a Minimal Inconsistent Subset

3.2.1 Basic Algorithm

A minimal inconsistent subset can be computed efficiently using a simple algorithm shown in Figure 3.2[1]. 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 $S = \{\}$. The self-inconsistent constraint *false* is used as a sentinel.

3.2.2 Improved Algorithm

A variant of the basic algorithm will compute a better minimal inconsistent subset. Let $C = \{c_1, \ldots, c_n\}$ be such that $i < j$ implies that the symbol (occurrence) imposing c_i occurs textually before the symbol (occurrence) imposing c_j. Then it is likely that a minimal inconsistent subset can be formed from a rather small range of the sequence $c_1, \ldots, c_n$, and such a local subset is considered a good explanation. If this is the case, scanning S in alternate directions will be more efficient and compute a better solution. An improved algorithm based on this consideration is shown in Figure 3.3.

[1]The algorithm described here is a revised version of the one proposed in [8] and takes into account the case when C is consistent.

```
c_{n+1} ← false;
S ← {};
i ← 0; j ← 1;
while S is consistent do
  D ← S;
  while D is consistent do
    i ← i + j; D ← D ∪ {c_i}
  end while;
  S ← S ∪ {c_i}; j ← −j
end while;
if i = n + 1 then S ← {} fi
```

Figure 3.3: Improved Algorithm for computing a minimal inconsistent subset

3.2.3 Cost

We consider the complexity of the basic algorithm in Figure 3.2. Although the algorithm above is quite general, its efficiency hinges upon the fact that there is a pair of efficient algorithms for computing the union of constraint sets and checking its consistency.

As explained in Section 2.2.4, it takes $O(nd \cdot \alpha(n))$ time to merge n mode constraints. The time complexity of finding a minimal subset with k elements out of n mode constraints is $O(nkd \cdot \alpha(n))$, because

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

From our experiments, k is usually a small value independent of the program size. 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 actual values of k are shown in Section 5.1.4.

The 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 the improved algorithm in Figure 3.3.

3.2.4 An Example

We consider a quicksort program with the confusion of two variables.

```
R1: quicksort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
R2: qsort([],     Ys0,Ys ) :- true | Ys =1 Ys0.
R3: qsort([X|Xs],Ys0,Ys3) :- true |
       part(X,Xs,S,L),qsort(S,Ys0,Ys1),
       Ys2 =2 [X|Ys1],qsort(L,Ys2,Ys3).
     (The body unification goal should have been Ys1 =2 [X|Ys2])
R4: part(_,[],     S, L ) :- true | S =3 [],L =4 [].
R5: part(A,[X|Xs],S0,L ) :- A>=X | S0 =5 [X|S],part(A,Xs,S,L).
R6: part(A,[X|Xs],S, L0) :- A< X | L0 =6 [X|L],part(A,Xs,S,L).
```

Either basic algorithm or improved algorithm returns the following minimal inconsistent subset of mode constraints:

	Mode constraint	Rule	Source symbol
(a)	$m(\langle \texttt{qsort}, 3\rangle) = \mathit{in}$	(BF)	"[]" in R_1
(b)	$m/\langle =_1, 1\rangle = m/\langle \texttt{qsort}, 3\rangle$	(BV)	Ys in R_2
(c)	$m/\langle =_1, 2\rangle = \overline{m/\langle =_1, 1\rangle}$	(BU)	$=_1$ in R_2
(d)	$m/\langle \texttt{qsort}, 2\rangle = m/\langle =_1, 2\rangle$	(BV)	Ys0 in R_2
(e)	$m(\langle =_2, 2\rangle) = \mathit{in}$	(BF)	"." in R_3
(f)	$m/\langle =_2, 2\rangle = \overline{m/\langle =_2, 1\rangle}$	(BU)	$=_2$ in R_3
(g)	$m/\langle =_2, 1\rangle = \overline{m/\langle \texttt{qsort}, 2\rangle}$	(BV)	Ys2 in R_3

This subset tells not only that the paths appearing above may have both mode values, *in* mode and *out* mode, but also why each path has been constrained to both mode values. For example, two inconsistent constraints can be derived from the subset:

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

After all, the source symbols that have imposed the constraints in the subset are suspicious. In this example, since the variables Ys1 and Ys2

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

Figure 3.4: Algorithm for computing multiple independent minimal inconsistent subsets

have wrongly been rewritten to each other, the constraint (g) is (one of) the culprit. Although `Ys1` has not been detected, the subset can identify the clauses R_1, R_2 and R_3 as the possible sources of the error. The other clauses R_4, R_5 and R_6 have been considered correct. Even if the definition of `quicksort` is a part of large programs, the subset can still locate the error.

3.3 Finding Multiple Independent Subsets

The two algorithms in Figures 3.2 and 3.3 compute a single minimal inconsistent subset S of C. To compute multiple, independent (disjoint) minimal inconsistent subsets, we can simply re-apply the algorithms after removing the elements of S from C. The algorithm for computing multiple, independent subsets at once is shown in Figure 3.4, where c_0 is a self-inconsistent constraint as a sentinel. Note that independent subsets found by this algorithm does not always correspond to different errors, because multiple subsets may indicate the same error (cf. Section 4.3).

3.4 Pinpointing Suspicious Constraints

Because we are dealing with near-misses, we can assume that most of the mode constraints obtained from a program represent an intended specification and that they have redundancies in most cases. In this case, one can often pinpoint a bug either

1. by computing a maximal consistent subset of size $n-1$ and taking its complement, or
2. by computing several overlapping minimal inconsistent subsets and taking their intersection.

To reduce the amount of computation, we do not compute all minimal subsets; instead, for each element (say s_i) of the initial inconsistent subset S, we compute a minimal inconsistent subset after removing s_i from C, which will lead to another minimal subset if it exists. Thus the two policies above can be combined into a single algorithm.

Let $S = \{s_1, \ldots, s_m\}$ be a minimal subset obtained by the algorithm either in Figure 3.2 or in Figure 3.3, and `getminimal`(C) be a function which computes a minimal inconsistent subset from a multiset C of constraints. Then, the algorithm that combines two policies of pinpointing is described in Figure 3.5, where T is a *multiset* of constraints what serves as counters of the numbers of constraints occurring in S and (various versions of) S', and $\dot{\cup}$ is a multiset union operator. The multiset T records how many times each constraint occurred in different minimal subsets. Under Policy 2, constraints with more occurrences in T are more likely to be related to the source of the error.

However, from our observations, these policies are not effective in either of two cases. First, if a program has low redundancy, the function `getminimal`$(C \setminus \{s_j\})$ cannot find another subset. Second, even if a program has high redundancy, multiple, independent subsets are often obtained even for a single error. As a result, another subset may not be obtained from a complementary set in which the independent subsets have been removed. Furthermore, an error may generate multiple wrong constraints, which also cause multiple, independent subsets to be obtained.

So, the policies are not used in the current version of Kima for efficiency, although the policies are not always ineffective. In reality, when either of the policies succeeds in refining suspicious constraints (for a single error),

$T \leftarrow S$;
for $j \leftarrow 1$ **to** m **do**
  $S' \leftarrow$ `getminimal`$(C \backslash \{s_j\})$;
  if $S' = \{\}$ **then**
    output $\{s_j\}$ as a solution of Policy 1
  else $T \leftarrow T \dot{\cup} S'$;
end for

Figure 3.5: Algorithm for pinpointing suspicious constraints

multiple, independent subsets are usually found for the error. Then, the search of alternatives is performed efficiently by Quick-check (Section 4.3) even without the policies.

Chapter 4

Automated Debugging

Here we explain the technique for searching alternatives to errors based on the suspected locations indicated by minimal inconsistent subsets. There are two problems to be solved for automated error correction. First, in general, multiple subsets are found independently, but one subset is not always corresponding to each independent error. For example, multiple subsets of mode and/or type constraints are often found for a single error. We have solved this problem by classifying the subsets into groups corresponding to individual errors. Second, multiple alternatives are usually found only with mode and type information. To refine the alternatives, we introduce plausibility criteria as syntactical constraints, by which we give priority on the alternatives. These two techniques, grouping and prioritizing, can be used also for improving the efficiency of searching alternatives. Also the locality of mode/type constraints contributes to the efficiency. We present optimized algorithms for automated error correction as well as the basic algorithm.

4.1 Target of 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 Figure 2.2).

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 have to choose a particular symbol to substitute, 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 would expand too much in trying to find both appropriate function and predicate symbols and their arguments.

We aim at a programming tool to assist the input process of programs into which a lot of trivial errors are prone to intrude. Unfortunately, our framework is not sufficient for the essential part of programming such as algorithm design, but mode and type system can be a guideline to develop reliable and readable programs.

4.2 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 and well-typed programs.

Given a possibly erroneous program L, which is a set $\{l_1, \ldots, l_n\}$ of

$w \leftarrow misv(L)$; $L_1 \leftarrow cls(w)$;
$L_0 \leftarrow L \setminus L_1$;
$S \leftarrow \{\}$;
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 4.1: Basic algorithm for automated error correction

clauses, let W_L be defined as

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

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 Figure 3.2 or 3.3.

The algorithm in Figure 4.1 finds a set S of alternative solutions to the program L, where $cls(w)$ ($w \subseteq W_L$) stands for the 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 ($\leq n$) 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 between alternatives by the depth at which they are found.

4.2.1 Cost

We consider the cost of the basic algorithm in Figure 4.1, namely depth-d search of alternatives from one minimal inconsistent subset. Let u be 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 ${}_{ur}\mathrm{C}_d \cdot {}_{t+d}\mathrm{P}_d$, which is the number of possible ways of rewriting d variable occurrences in clauses indicated by a minimal inconsistent subset. ${}_{ur}\mathrm{C}_d$ is concerned with the choice of variable occurrences to be rewritten, whereas ${}_{t+d}\mathrm{P}_d$ is the maximum number of ways of rewriting d 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 the product of the number of programs generated in the search and the cost of mode and type analyses:

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

The average sizes of r and t are 13.4 and 6.58, respectively, for the case of the Kima system, which is a KL1 program of 5,200 lines long. For most

programs with about 10 variable occurrences in a clause, u is smaller than 4 from our experiences. For programs with more than 10 variable occurrences in a clause, rather large subsets are often found. We will discuss the value of u for large programs in Section 5.1.4.

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 (Section 5.1.3). We can take advantage of this fact, as well as Quick-check, to save the number of mode and type analyses (Sections 4.5.1 and 4.5.2).

4.3 Grouping Errors

As we mentioned in Section 3.3, 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 together minimal inconsistent subsets sharing the same clause (Figure 4.2). A group thus formed plays the role of a unit of searching alternatives to erroneous clauses.

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

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

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-d 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

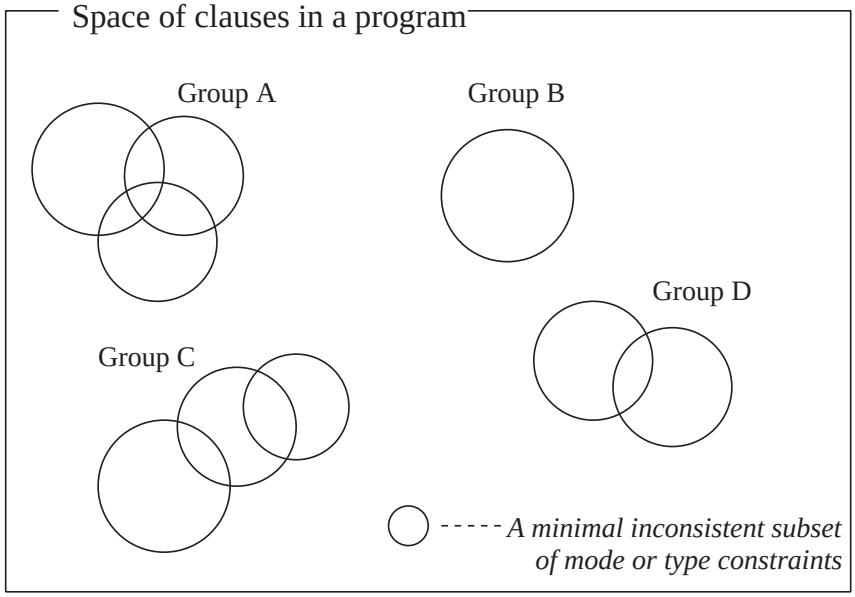

Figure 4.2: Grouping minimal inconsistent subsets

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 comprises 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 Figure 4.3.

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.

4.4 Constraints Other Than Modes and Types

4.4.1 Prioritizing Alternatives

Kima searches alternative solutions using mode and type information, but multiple alternatives are found in many cases. Kima refines the quality of

```
for i ← 1 to n do
  L_i ← ⋃_{j=1}^{g_i} cls(w_ij);
end for;
L_0 ← L \ ⋃_{j=1}^{n} L_j;
for i ← 1 to n do
  S_i ← {};
  for d ← 1 to d_MAX do
    for each q ∈ Q_{L_i}^d do
      if quick_check(q, i) then
        if mcs(L_0 ∪ q) is consistent ∧ tcs(L_0 ∪ q) is consistent then
          S_i ← S_i ∪ {q}
        fi
      fi
    end for
  end for
end for

function quick_check(q, i)
  return ⋀_{j=1}^{g_i}(w_ij ∩ mut(q) ≠ {})
end.
```

Figure 4.3: Algorithm for automated error correction with grouping

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* (Section 2.2)[1] 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 the constraint $\alpha = list(\alpha)$ imposes a strong type constraint on α and is therefore unlikely.

Kima prioritizes multiple alternatives by imposing penalty points on unlikely symbol occurrences. An alternative with lower total penalty points has a higher priority. The parameters of penalty points adjusted in Kima

[1]Variable occurrences falling under Heuristic Rule 1.3 do not directly impose *IN* or *OUT*, but mode constraints imposed by such variable occurrences are strengthened in many cases (Section 2.2.1).

Table 4.1: Penalty points imposed on unlikely variable occurrences

Variable occurrences in Heuristic Rule 1	Penalty points
1. singleton occurrence	
- If the variable name does not begin with an underscore "_"	
* If the variable is in a clause head	2
* If the variable is in a clause body	3
- If the variable name begins with an underscore"_"	1
2. two occurrences in a clause head	2
(for one more occurrence)	1
3. three occurrences in the head and/or the body of a clause	1
(for one more occurrence)	1
4. two occurrences as arguments of the same body goal	2
(for one more occurrence)	2

for Heuristic Rule 1 are in Table 4.1. Heuristic Rule 2 does not impose penalty points but lower the priority by one (Section 4.5.1).

4.4.2 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 checked for its value or instantiation 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 "_".

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 the 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.

4.5 Optimizing Search of Alternatives

In addition to Quick-check (Section 4.3), prioritizing with Heuristic Rules (Section 4.4.1) and Detection Rules (Section 4.4.2) are also effective for the number of mode and type analyses to be reduced in generate-and-test search. Applying these two techniques before performing the test with mode and type analyses has an effect not only on the test but also on the generation of mutated clauses. Furthermore, the cost of inevitable mode and type analyses can be reduced by taking advantage of the locality of mode and type constraints.

4.5.1 Optimization of Test

The algorithm shown in Figure 4.4 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 ob-

```
for i ← 1 to n do
  L_i ← ⋃_{j=1}^{g_i} cls(w_ij);
end for;
L_0 ← L \ ⋃_{j=1}^{n} L_j;
for i ← 1 to n do
  for k ← 1 to k_MAX + 1 do
    S_{i,k} ← {}
  end for;
  for d ← 1 to d_MAX do
    prioritize Q^d_{L_i} into Q_1, ..., Q_{h_i} with Heuristic Rule 1;
    j ← 1;  k ← 1;
    while j ≤ h_i ∧ k ≤ k_MAX do
      for each q ∈ Q_j do
        if quick_check(q, i) then
          if detection_rules_ok(q) then
            if mcs(L_0 ∪ q) is consistent
               ∧ tcs(L_0 ∪ q) is consistent then
              if heuristic_rule2_ok(q) then
                S_{i,k} ← S_{i,k} ∪ {q}
              else
                S_{i,k+1} ← S_{i,k+1} ∪ {q}
              fi
            fi
          fi
        fi
      end for;
      if S_{i,k} ≠ {} then
        k ← k + 1
      fi;
      j ← j + 1
    end while
  end for
end for
```

Figure 4.4: Optimized algorithm 1 for automated error correction

serves 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.

4.5.2 Optimization of Generation

Outline of an Algorithm

From our experiments, it proved to take much time and memory to generate all programs in the set $Q^d_{L_i}$, but not all programss are necessary when we are interested only in high-priority alternatives. Since the generation of mutated programs itself is a search problem, unnecessary programs can be not generated at all by pruning low-priority mutations during the search. The current version of Kima hence generates mutated programs with high priority on demand by the prioritizing with Heuristic Rule 1 and by the check with Detection Rule 1.

Let *high_priority*(L, h, d) be the function that generates sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_h$ of high-priority mutated clauses where at most d variable occurrences have been mutated from the clauses L. Then the whole algorithm with the optimization of the generation is described as in Figure 4.5, where the constant MAX_h is for termination.

The function *detection_rule2_ok*(L) returns *true* iff the program L observes Detection Rule 2 and a command line option is specified to employ the rule (Section A.2). Unless the option is specified, the function always returns *true*. The employment of Detection Rule 2 is optional, whereas that of Detection Rule 1 is indispensable. However, in the generation of mutated clauses, prioritizing with Heuristic Rule 1.1 efficiently prunes low-prirority clauses with singleton variables instead of Detection Rule 2.

Generating High-Priority Clauses

Among clauses indicated by minimal inconsistent subsets, there are clauses actually including errors and clauses including no error. In many cases, errors in a clause corrupt the penalty points of the clause. So the function *high_priority*(L, h, d) generates sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_h$ of high-priority mutated clauses by combining high-priority clauses mutated from each clause in L.

Let $L = \{l_1, l_2, \ldots, l_n\}$, and $\mathcal{L}^i_{h,d}$ be the set of priority-h clauses generated by mutating d variable occurrences from a clause $l_i \in L$. The union

```
for i ← 1 to n do
  L_i ← ⋃_{j=1}^{g_i} cls(w_ij);
end for;
L_0 ← L \ ⋃_{j=1}^{n} L_j;
for i ← 1 to n do
  for k ← 1 to k_MAX + 1 do
    S_{i,k} ← {}
  end for;
  h ← 1;  k ← 1;
  while k ≤ k_MAX ∧ h ≤ MAX_h do
    h ← k_MAX − k + h;
    Q_1, ..., Q_h ← high_priority(L_i, h, d_MAX);
    for j ← 1 to h do
      for each q ∈ Q_j do
        if quick_check(q, i) then
          if detection_rule2_ok(q) then
            if mcs(L_0 ∪ q) is consistent
               ∧ tcs(L_0 ∪ q) is consistent then
              if heuristic_rule2_ok(q) then
                S_{i,k} ← S_{i,k} ∪ {q}
              else
                S_{i,k+1} ← S_{i,k+1} ∪ {q}
              fi
            fi
          fi
        fi
      end for;
      if S_{i,k} ≠ {} then
        k ← k + 1
      fi
    end for
  end while
end for
```

Figure 4.5: Optimized algorithm 2 for automated error correction

$\bigcup_{d=0}^{d_{MAX}} \mathcal{L}^i_{1,d}$ is the set of clauses with the least penalty points in $\bigcup_{d=0}^{d_{MAX}} Q^d_{\{l_i\}}$, and must not be $\{\}$. In contrast, the union $\bigcup_{d=0}^{d_{MAX}} \mathcal{L}^i_{h,d}$ for $h \geq 2$ is a set of clauses with penalty points more than the least by $h-1$, and may be $\{\}$. Then, $\mathcal{Q}_i$ is defined as:

$$\mathcal{Q}_i = \{\{l_1, \ldots, l_n\} |\ l_1 \in \mathcal{L}^1_{h_1,d_1}, \ldots, l_n \in \mathcal{L}^n_{h_n,d_n},\ \sum_{j=1}^{n} h_j = n+i-1,\ \sum_{j=1}^{n} d_j \leq d_{MAX}\}$$

Note that 0-mutated clause is the original clause with no mutation.

During the search/generation of mutated clauses, it is necessary to prune nodes/clauses that will not be able to mutate into clauses in $\mathcal{L}^i_{h,d}$ by checking both the improvability of priority and the observability of Detection Rule 1. One mutation of a variable occurrence can be thought of as a pair of a *pickup* and a *putdown* operations. Let U_l be a set of clauses obtained by picking up a variable occurrence in the clause l. That is, a variable occurrence is ignored in $u \in U_l$. Let D_u be a set of clauses obtained by putting down a different variable in place of the variable occurrence that has been picked up in u. Then, given a clause l_i, the algorithm in Fig 4.6 computes $\mathcal{L}^i_{h,d}$ for priority h ($1 \leq h \leq h_{MAX}$) and depth d ($0 \leq d \leq d_{MAX}$).

The algorithm does not generate mutated clauses with penalty points more than the constant MAX_{pp}, which is set, by default, to 20 for termination (see the argument of the option `+pp` in Section A.2). The constant MAX_h in Fig 4.5 is set to the same value for simplicity. By combining clauses in each $\mathcal{L}^i_{h,d}$ ($1 \leq i \leq n$), the function $\mathit{high\_priority}(L, h_{MAX}, d_{MAX})$ generates the sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_{h_{MAX}}$.

The improvability of penalty points and the observability of Detection Rule 1 are checked after every pickup and putdown operation. The functions $\mathit{reachable\_mid}(l, d, pp)$ and $\mathit{reachable}(l, d, pp)$ return *true* iff the clause l has both possibilities after a *pickup* and a *putdown*, respectively, for the current depth d and penalty points pp. The function $\mathit{found}(l, pp)$ returns *true* iff the clause l observes Detection Rule 1 and has the penalty points pp.

Let $\mathit{num\_violate\_dr11}(l)$ and $\mathit{num\_violate\_dr12}(l)$ represent the number of (not variable occurrences but) variables that violate Detection Rule 1.1 and Detection Rule 1.2, respectively, and $\mathit{penalty\_point}(l)$ represent penalty points the clause l has. Note that these three functions can also be applied to $u \in U_l$, because they check only the number and the positions of variable occurrences. Then, the function $\mathit{found}(l, pp)$ is defined as:

$$\mathit{found}(l, pp) \quad = \quad \mathit{num\_violate\_dr11}(l) = 0 \wedge \mathit{num\_violate\_dr12}(l) = 0$$

$h \leftarrow 1;\ pp \leftarrow 0;$
while $h \leq h_{MAX}\ \wedge\ pp \leq MAX_{pp}$ **do**
  $d \leftarrow 0;\ T \leftarrow \{\};$
  if $reachable(l_i, d, pp)$ **then**
    $T \leftarrow \{l_i\};$
    if $found(l_i, pp)$ **then**
      $\mathcal{L}^i_{h,d} \leftarrow \{l_i\}$
    fi
  fi;
  for $d \leftarrow 1$ **to** d_{MAX} **do**
    $T' \leftarrow \{\};\ \mathcal{L}^i_{h,d} \leftarrow \{\};$
    for each $l \in T$ **do**
      for each $u \in U_l$ **do**
        if $reachable\_mid(u, d, pp)$ **then**
          for each $l' \in D_u$ **do**
            if $reachable(l', d, pp)$ **then**
              $T' \leftarrow T' \cup \{l'\};$
              if $found(l', pp)$ **then**
                $\mathcal{L}^i_{h,d} \leftarrow \mathcal{L}^i_{h,d} \cup \{l'\}$
              fi
            fi
          end for
        fi
      end for
    end for
  end for;
  if $h > 1$ **then**
    $h \leftarrow h + 1$
  else if $\bigcup_{d=0}^{d_{MAX}} \mathcal{L}^i_{h,d} \neq \{\}$
    $h \leftarrow 2$
  fi;
  $pp \leftarrow pp + 1$
end while

Figure 4.6: Algorithm for generating high-priority clauses

$$\wedge\, penalty\_point(l) = pp.$$

An Example of the Evaluation Functions of A* Search

The algorithm in Fig 4.6 is an iterative-deepening A* search with respect to penalty points (priority), where $reachable\_mid(l, d, pp)$ and $reachable(l, d, pp)$ are the evaluation functions for pruning. In the current version of Kima, $reachable(l, d, pp)$ is defined as:

$$\begin{aligned} reachable(l, d, pp) &= (num\_violate\_dr11(l) + num\_violate\_dr12(l) \leq d_{rest}) \\ &\quad \wedge (penalty\_point(l) \leq pp + 2 \cdot 3 \cdot d_{rest}), \end{aligned}$$

where $d_{rest} = d_{MAX} - d$. If, for $u \in U_l$, a variable occurrence picked up is in a clause head and $num\_violate\_dr11(u) \geq 1$, then $reachable\_mid(u, d, pp)$ is defined as:

$$\begin{aligned} reachable\_mid(u, d, pp) &= (num\_violate\_dr11(u) + num\_violate\_dr12(u) \\ &\quad \leq d_{rest} + 1) \\ &\quad \wedge (penalty\_point(u) \leq pp + 2 \cdot 3 \cdot d_{rest} + 3); \end{aligned}$$

otherwise

$$\begin{aligned} reachable\_mid(u, d, pp) &= (num\_violate\_dr11(u) + num\_violate\_dr12(u) \\ &\quad \leq d_{rest}) \\ &\quad \wedge (penalty\_point(u) \leq pp + 2 \cdot 3 \cdot d_{rest} + 3). \end{aligned}$$

The parameters in these definitions are adjusted from two considerations. First, for Detection Rule 1, the pickup operation of a variable occurrence either in a clause head or in a clause guard may decrease $num\_violate\_dr11(l)$ by one, whereas that in a clause body may decrease $num\_violate\_dr12(l)$ by one. Also, the putdown operation of a variable occurrence in a clause head may decrease $num\_violate\_dr11(l)$ by one. Accordingly depth-d search (the mutation of d variable occurrences) may decrease the sum of $num\_violate\_dr11(l)$ and $num\_violate\_dr12(l)$ by d. Second, for penalty points, either a pickup or a putdown operation may decrease the penalty points of a clause by three (see Table 4.1). A variable occurrence can possibly fall under both Heuristic Rules 1.2 and 1.3 simultaneously or under both Heuristic Rules 1.3 and 1.4, but even such a variable imposes at most three penalty points.

(Penalty points are thus adjusted cooperatively with the evaluation functions.) Accordingly depth-d search may decrease the penalty points of a clause by $2 \cdot 3 \cdot d$. Though more precise evaluation functions might of course be defined, it is the problem of tradeoff between efficiency and the conciseness of definition.

Equivalence of Programs

After the execution of the algorithm in Fig 4.6, with respect to the equivalence of clauses, Kima eliminates duplicate clauses in $\bigcup_{d=0}^{d_{MAX}} \mathcal{L}_{h,d}^{i}$ generated from the clause l_i for each priority h. Clauses (programs) are *equivalent*

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

Out of equivalent clauses, one of clauses with the smallest d (i.e., the number of mutations) is left, and the other equivalent clauses are all eliminated. Different numbers of the mutations of a clause may generate equivalent clauses.

Programs with different penalty points (priority) are not considered equivalent even if the programs are equivalent according to the two conditions above. All this intention is to discriminate between a program including a singleton variable whose name begins with an underscore "_" and a program including, at the same position, a singleton variable whose name does not.

For a quicksort program containing two wrong variable occurrences in the same clause (Section 5.2.3), the above optimization of both the generation and the test improved the response time of computing the highest-priority alternatives from 7.78 seconds to 0.37 seconds on the KLIC system running on a PC/AT compatible machine with PentiumIII 733 MHz and 768 MB of memory.

4.5.3 Optimization Using the Locality of Mode and Type Constraints

Checking modes and types of a rewritten program requires the cost proportional to the program size (Section 2.2.4), but inconsistency usually occurs

within a small region of program text (Section 3.2.3). The performance of the test of mutated programs is therefore improved by analyzing those constraints imposed by *suspected neighborhood*, that is, the suspected predicates and predicates closely related to them with respect to caller-callee relationship before taking other constraints into account.

We considered two kinds of suspected neighborhoods to be analyzed in advance, and compared them:

1. all the definition clauses of predicates (say L) including a clause indicated by a group of minimal inconsistent subsets, and

2. in addition to L, the definition clauses of both predicates calling a predicate in L and predicates called from L.

The clauses indicated directly by other groups of minimal inconsistent subsets are removed from the suspected neighborhood, because those clauses may contain errors.

The region in the second way is definitely larger than that in the first way. This chain of predicate call relation can be traced at any number of steps until the suspected neighborhood covers the whole program, but the current version of Kima employed the first way from experiments in Section 5.1.4.

Chapter 5

Experiments and Examples

We show four experiments with which we discuss both the effectiveness and the ability of the Kima system. We then explain the details of our framework by illustrating the behavior of Kima based on three examples.

5.1 Experiments

In the first three experiments, we investigated how many of programs with a couple of errors were detected as erroneous by Kima, how many alternatives it proposed given an erroneous program, and how many "plausible" programs there were in the neighborhood of a correct (original) program. Sample programs we used here are list concatenation (`append`), the generator of a Fibonacci sequence (`fibonacci`), 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.

In the last experiment, we investigate the current ability of Kima based on other sample programs of non-trivial size. Simultaneously, we do not only discuss the effectiveness of the local analysis (Section 4.5.3) but also compare the two ways of delimiting suspected neighborhoods.

5.1.1 Single Error Detection and Correction

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

Table 5.1: Single-error detection and correction

Program	Analysis	Level	Prioritizing	Total cases	Detected	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

how many alternatives were reported. Table 5.1 shows the results[1]. 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.

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

[1]In a similar experiment shown in our early paper [3], 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.

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. Some variable occurs four times in the correct `fibonacci` program (`N2` in the program in Section 5.2.2), 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.

5.1.2 Error Detection Rate

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 5.2 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 (Section 4.5.2).

When multiple errors exist in some clause of a program, the error is detected as long as at least one of the errors cause 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.

5.1.3 The Number of Plausible Programs

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 the programs have the same (or higher) priority as the original program. The result is shown in Table 5.3. 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 our plausibility criteria looked like.

In contrast to the experiments in Section 5.1.1 and 5.1.2 (and the next 5.1.4), here we included the mutation of variables to those whose names began with "_" in order to precisely count the number of plausible programs. In the column "Plausible programs", equivalent programs (Section 4.5.2) were counted as one program, and the original, intended programs were not counted.

Table 5.2: 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

Table 5.3: The number of plausible programs in the programs with N mutations

Program	N	Total cases	Plausible programs	With caller info	Unrelated to guard
`append`	1	58	0	0	0
	2	1200	5	1	1
	3	16980	14	3	3
	4	167842	27	5	5
`fibonacci`	1	118	10	9	5
	2	4668	64	3	0
	3	133045	307	5	0
`quicksort`	1	300	8	4	0
	2	12102	27	17	13
	3	337455	72	48	32

From Table 5.3 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 alternatives proposed by Kima 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 5.3.

Caller information (i.e., examples of clauses containing calls to the predicates to be debugged) can reduce the number of plausible programs, because it plays the role of mode and type specifications. The column "With caller info" indicates the number of plausible programs in the existence of caller information. Out of those programs, programs whose guard goals have not essentially been rewritten are indicated in the column "Unrelated to guard". That is, the rightmost column shows the number of mutated but plausible programs in which neither (i) a variable tested in a clause guard was mutated, nor (ii) a variable tested in a clause guard was made to occur at a different argument position in the clause head.

In programs "Unrelated to guard", while more than half of those programs diverge or cause deadlock, the other programs return unintended output. For example, we found:

1. a program that merges two input lists by taking their elements alternately in the neighborhood of `append` (see Example 1 in Section A.1),

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.

However, most of the programs were such that we could not give any concise meaning to their behavior and output.

5.1.4 Search Space Reduced by the Local Analysis of Suspected Neighborhood

We investigated how mutated programs generated in the search of alternatives would be sieved through the mode and type analyses of suspected neighborhoods and through the analyses of the whole program. Sample programs we used here are `nqueen` (of 67 lines long) and three modules of Kima itself, `gen_test` (of 200 lines long), `tgraph` (of 218 lines long) and `graph` (of 390 lines long); the program `nqueen` computes the number of the solutions of N-queen problem in parallel; the module `gen_test` is the top module that calls two modules for the generation and the test of alternatives, and is the version where the second way about suspected neighborhoods was implemented; the module `tgraph` is the solver of type constraints, while the module `graph` is the solver of mode constraints.

For each sample program, we randomly generated 100 cases of programs into which one or two mutations of variable occurrences in the same clause were inserted, and applied depth-1 or depth-2 search of alternatives, respectively. Here, we used two versions of Kima that implemented the two ways of delimiting suspected neighborhoods (Section 4.5.3). We then compared how many of mutated programs generated in the search were well-moded and well-typed within a suspected neighborhood. The result is shown in Tables 5.4 and 5.5. Here, error detection was performed at detection level 2. The experiment was performed on a PC/AT compatible machine with PentiumIII 733 MHz and 768 MB of memory. Although there were several predicate calls of other modules in the sample modules of Kima, callee side information was not used. Nevertheless it did not cause problem to the quality of error correction.

In Table 5.4, the column "Canceled cases" indicates the number of cases where errors were detected but their correction was canceled (impossible) for either of two reasons. One is the occurrence of memory swap, and the other is that the error correction took over twenty minutes. The column "Generated programs" indicates the average number of high-priority programs generated by the function $high\_priority(L, h, d)$ with the algorithm in Figure 4.6 except the canceled cases. The column "Passed local analysis" indicates the average number of programs that were well-moded and well-typed within a suspected neighborhood delimited in the two ways. The column "Final alternatives" indicates the average number of programs pro-

Table 5.4: Effect of local analysis for the programs with N mutations

Program	N	Detected cases	Canceled cases	Generated programs	Passed local analysis		Final alternatives
					first	second	
nqueen	1	96	0	2.45	1.96	1.89	1.10
	2	96	1	53.0	27.7	20.0	2.17
gen_test	1	99	0	2.45	2.05	1.94	1.67
	2	83	3	60.6	40.2	21.5	5.74
tgraph	1	100	0	2.42	1.82	1.68	1.21
	2	100	37	28.4	19.2	13.3	2.35
graph	1	98	1	1.98	1.40	1.33	1.15
	2	97	39	24.5	10.9	8.14	2.28

Table 5.5: Average response time of Kima with local analysis

Program	Clauses	Variable occurrences	Kinds of variables	N	Response time (sec)	
					first	second
nqueen	34	10.4	5.12	1	0.439	0.481
				2	19.1	19.2
gen_test	77	11.6	5.84	1	2.04	2.10
				2	16.5	16.1
tgraph	79	17.6	8.84	1	10.9	11.9
				2	60.4	61.8
graph	155	16.8	8.46	1	21.7	25.1
				2	87.4	91.8

posed as alternatives by the mode and type analyses of the whole program. The average response time of Kima except the canceled cases is shown in Table 5.5. The number of clauses, the average number of variable occurrences in a clause, and the average number of different variables in a clause are shown also. From the column "Passed local analysis" in both Tables, the two ways of delimiting suspected neighborhoods turned out not to make much difference to the efficiency of the search. So the current version of Kima employed the first (simpler) way.

In most cases, the number of final alternatives was one or quite small even for errors in the programs of rather long lines. The correction of a program with two mutations does not always need depth-2 search because of the equivalence of programs. This is why the number of detected cases in the row with `gen_test` and $N = 2$ was relatively small. The mutations of variable occurrences may rewrite a program into its equivalent program. Since `gen_test` has more than ten clauses with at most only four variable occurrences, two mutations of such clause are likely to make an equivalent program. There were also cases where an excessive mutation for the search increased the number of alternatives finally reported. In the row with `gen_test` and $N = 2$, there was a case where 184 alternatives were accidentally found for a clause with two out of 25 variable occurrences mutated. If we exclude this case, the average number in the column "Final alternatives" is improved to 3.44.

The Limit of Kima's Abilities and Its Solution

In canceled cases, the reason why a large amount of the resource of time and/or space was consumed is that a lot of clauses were indicated by a group of minimal inconsistent subsets. The cost of the error correction is sensitive to the number of the suspected clauses (the value of u in Section 4.2.1).

So, how many suspected clauses is the limit of error correction not canceled? In the row with `graph` and $N = 1$, there was only a case where depth-1 search was canceled. In this case, four minimal inconsistent subsets (of located variables) were found; two subsets on modes had 7 and 25 elements, respectively, and the other two subsets on types had 20 and 25 elements. The four subsets indicated 29 clauses in total. In the canceled cases with $N = 2$, a group of subsets indicated more than ten (often twenty) clauses.

For programs with a number of variable occurrences included in a clause, the size of minimal inconsistent subsets was prone to be large. When an error is in a predicate, the size of a subset depends on the number of both predicates calling the erroneous predicate and predicates called from the erroneous predicate. This number is then related to the number of variable occurrences in a clause. Also, the size of a subset of type constraints was usually larger than that of mode constraints, when the subsets of both modes and types were found at a time. A minimal inconsistent subset of type constraints traces between two function occurrences of different types, but the two occurrences that cause inconsistency are often far from each other. The number of the occurrences of function symbols are quite small compared to all symbol occurrences in a program, whereas the two mode values (*in* and *out*) are ubiquitous over any clause, predicate and program.

A lot of clauses are suspected by a group of subsets because

1. one of the subsets consists of a lot of constraints indicating a lot of clauses, and/or

2. several subsets classified into a group amount to indicate a lot of clauses.

In most canceled cases, both situations were observed simultaneously, but one of the subsets in a group had less than 10 constraints. Therefore, suppose errors are in the intersection of the subsets of the same group, error correction that was canceled might be completed. This assumption is true as for this experiment in which errors were inserted into the same clause, but is not considered in the current version of Kima.

5.2 Examples

We illustrate three examples of automated error correction. The first two are the correction of a single error. By these examples, we explain the grouping and the prioritizing. The last one is the correction of two interdependent errors, and is a little advanced example including the correction of a variable occurrence to a constant symbol.

5.2.1 Append Program With an Error

We discuss an `append` program with a single error. This example is simple and yet instructive.

R_1: `append([], Y,Z ) :- true |` $\mathtt{Y}=_1\mathtt{Z}$.
R_2: `append([A|Y],Y,Z0) :- true |` $\mathtt{Z0}=_2$`[A|Z], append(X,Y,Z).`
(The head should have been `append([A|X],Y,Z0)`)

Either algorithm in Figure 3.2 or 3.3 computes the following minimal inconsistent subset of mode constraints:

Mode constraint	Rule	Source symbol
$m/\langle \mathtt{append}, 1\rangle\langle ., 2\rangle = IN$	(HV)	`Y` in R_2
$m/\langle \mathtt{append}, 1\rangle = OUT$	(BV)	`X` in R_2

Here, we do not consider Detection Rule 2, though it can detect the variable `X` in Clause R_2 as an error.

This tells that we should suspect the variables `X` and `Y` in Clause R_2. Because the variable `X` has wrongly been mutated into the variable `Y`, this indication is legitimate. A symbol either with more occurrences or with less occurrences caused by the wrong mutation is likely to be detected, but both symbols are detected in this case.

Then, depth-1 search tries to rewrite each variable occurrence in two ways; one way is to rewrite the variable `X` or `Y` to the other variables in Clause R_2; the other way is to rewrite the other variables to the variable `X` or `Y`. Without prioritizing, the search finds six well-moded and well-typed alternatives:

(1) R_2: `append([A|X],Y,Z0) :- true |` $\mathtt{Z0}=_2$`[A|Z], append(X,Y,Z).`

(2) R_2: `append([A|Y],X,Z0) :- true |` $\mathtt{Z0}=_2$`[A|Z], append(X,Y,Z).`

(3) R_2: `append([A|Y],Y,Z0) :- true |` $\mathtt{Z0}=_2$`[A|Z], append(Z0,Y,Z).`

(4) R_2: `append([A|Y],Y,Z0) :- true |` $\mathtt{Z0}=_2$`[A|Z], append(A,Y,Z).`

(5) R_2: `append([A|Y],Y,Z0) :- true |` $\mathtt{Z0}=_2$`[A|Z], append(Z,Y,Z).`

(6) R_2: `append([A|Y],Y,Z0) :- true |` $\mathtt{Z0}=_2$`[A|Z], append(Y,Y,Z).`

Types do not help much in this example, although Alternative (5) is given a low priority by Heuristic Rule 2 with respect to types (Section 4.4.1). Further, if the elements of the input lists received by `append` were not of list type on the caller side (e.g., `append([1,2,3],[4,5,6],Out)`) and this information was available, Alternative (5) would have been eliminated. Additionally, Heuristic Rule 1 imposes penalty points on Alternatives (3), (4), (5) and (6). Heuristic Rule 1.1, 1.2 and 1.3 are applied to all of them, and Heuristic Rule 1.4 are applied to Alternatives (5) and (6). In reality, these four alternatives are programs that cause reduction failure for most input data because of the two occurrences of the variable `Y` in the clause heads.

What are Alternatives (1) and (2) with the highest priority (with no penalty point)? Alternative (1) is the intended program, and Alternative (2) is a program that merges two input lists by taking their elements alternately. It's not '`append`', but is a quite meaningful program compared with the other alternatives! The actual output of Kima for this error can be found in Section A.1.

The optimization technique of generating mutations (Section 4.5.2) efficiently generates only high-priority mutations (1) and (2) both by making less occurrences of `Y` and by making more occurrences of `X` simultaneously. Although the optimization is not so important in this example, the generation of mutated programs is itself costly for deep search.

5.2.2 Fibonacci Sequence Program With an Error

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 computes three independent minimal inconsistent subsets; two on modes and one on types.

Minimal inconsistent subset 1 (on modes):

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

Minimal inconsistent subset 2 (on modes):

Mode constraint	Rule	Source symbol
$m(\langle \texttt{=}_2, 2\rangle) = in$	(BF)	"`.`" in R_2
$m/\langle \texttt{=}_2, 2\rangle = \overline{m/\langle \texttt{=}_2, 1\rangle}$	(BU)	$\texttt{=}_2$ in R_2
$m(\langle \texttt{=}_2, 1\rangle) = IN$	(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_τ)	`N1` in R_2
$\tau(\langle \texttt{=}_2, 2\rangle) =$ list type	(HBF_τ)	"`.`" in R_2
$\tau/\langle \texttt{fib}, 2\rangle = \tau/\langle \texttt{=}_2, 1\rangle$	(HBV_τ)	`N1` in R_2
$\tau/\langle \texttt{=}_2, 2\rangle = \tau/\langle \texttt{=}_2, 1\rangle$	(BU_τ)	$\texttt{=}_2$ in R_2
$\tau(\langle \texttt{:=}, 2\rangle\langle \texttt{+}, 1\rangle) =$ integer type	built-in	`:=` in R_2

These three subsets are classified into the same group because all the subsets indicate 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 Clause R_1 need not be considered, because such changes may dissolve the subset 1 but neither the subset 2 nor 3. After all, Quick-check finds that the only possible ways to dissolve all inconsistencies are either to replace `Ns0` by `N1` or vice versa in Clause R_2. Since 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 mutations would have had to be attempted in the search.

In this example, Kima finally finds only one alternative, which is the program we have intended. If we consider Detection Rule 2, it detects the variable `Ns0` in the clause head of R_2 as an error. This variable is dealt with

as the fourth minimal inconsistent subset with one element. In this case, the table of suspected variable symbols is as follows:

Clause	Variable symbol	Subset number
R_1	`Ns0`	1
R_2	`Ns0`	1, 4
R_2	`N1`	2, 3

This leads to the same process of the search as for this example. However, Detection Rules provably reduce the cost of searching alternatives as well as reinforce the detection power.

5.2.3 Quicksort Program With Two Errors

We consider a little advanced error correction by using a quicksort program with two interdependent errors, which is the same example as in Section 3.2.4. We consider the correction of constant symbols as well as the correction of variable occurrences by the similar technique.

```
1: R1: quicksort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
2: R2: qsort([],    Ys0,Ys ) :- true | Ys =1 Ys0.
3: R3: qsort([X|Xs],Ys0,Ys3) :- true |
4:         part(X,Xs,S,L),qsort(S,Ys0,Ys1),
5:         Ys2 =2 [X|Ys1],qsort(L,Ys2,Ys3).
     (The body unification goal should have been Ys1 =2 [X|Ys2])
```

Here, the definition clauses of the predicate `part` are omitted.

We re-present the minimal inconsistent subset of mode constraints obtained from the erroneous `quicksort` program:

Mode constraint	Rule	Source symbol
$m(\langle \texttt{qsort}, 3\rangle) = in$	(BF)	"`[]`" in R_1
$m/\langle \texttt{=}_1, 1\rangle = m/\langle \texttt{qsort}, 3\rangle$	(BV)	`Ys` in R_2
$m/\langle \texttt{=}_1, 2\rangle = \overline{m/\langle \texttt{=}_1, 1\rangle}$	(BU)	$\texttt{=}_1$ in R_2
$m/\langle \texttt{qsort}, 2\rangle = m/\langle \texttt{=}_1, 2\rangle$	(BV)	`Ys0` in R_2
$m(\langle \texttt{=}_2, 2\rangle) = in$	(BF)	"`.`" in R_3
$m/\langle \texttt{=}_2, 2\rangle = \overline{m/\langle \texttt{=}_2, 1\rangle}$	(BU)	$\texttt{=}_2$ in R_3
$m/\langle \texttt{=}_2, 1\rangle = \overline{m/\langle \texttt{qsort}, 2\rangle}$	(BV)	`Ys2` in R_3

We consider the correction of both constants and variables here, but do not consider Detection Rules for your information. Depth-1 search is first

tried but cannot find any alternative to the errors. Then, in the following depth-2 search, one mutation is performed on the possible sources of errors above, but the other mutation is attempted exhaustively in Clauses R_1, R_2 and R_3. Finally, however, the search successfully finds six well-moded and well-typed alternatives:

(1) Line 1: `quicksort(Xs,Ys) :- true | qsort(Xs,Zs,Zs).`

(2) Line 1: `quicksort(Xs,Ys) :- true | qsort(Zs,Ys,Zs).`

(3) Line 1: `quicksort(Xs,Ys) :- true | qsort(Xs,`c`,Ys).`

(4) Line 1: `quicksort(Xs,Ys) :- true | qsort(`c`,Ys,Xs).`

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

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

Here, c is some constant.

Alternatives (3), (4) and (6) have the highest priority, while (1), (2) and (5) have low priority due to Heuristic Rule 1.4. Heuristic Rule 1.1 imposes further penalty points on Alternatives (1) and (2). Alternative (5) causes occur-check error (Detection Rule 1.2), which is a (strong) kind of the constraints of Heuristic Rule 1.4.

If usage information is available, that is, if we know that `quicksort` is used as $m(\langle \texttt{quicksort}, 1\rangle) = \mathit{in}$ and $m(\langle \texttt{quicksort}, 2\rangle) = \mathit{out}$, Alternatives (1), (2) and (4) are excluded.

Of the remaining, Alternative (6) is the intended program that sorts items in ascending order. It is interesting to see that Alternative (3) is a program for sorting items in *descending* order by choosing '`[]`', the simplest element of list type, as the constant c. This is not an intended program, but is a reasonable and approximately correct alternative which should not be rejected in the absence of program specification. The current version of Kima finds nothing but Alternative (6) (Example 3 in Section A.1).

Chapter 6

Conclusions

6.1 Our Framework and the Kima System

We proposed a framework for automated (systematic) debugging with the strong support of mode and type system, static analysis, and constraint satisfaction. The framework comprises the techniques below:

- locating errors by computing minimal inconsistent subsets of mode/type constraints and by grouping the subsets,
- searching alternatives to an erroneous program by means of a pair of the generation of mutated programs and the test of the programs with mode/type analysis,
- enhancing the quality of alternatives by heuristic rules prescribing the (syntactical) "plausibility" of programs, and
- optimizing the generate-and-test search by using both the iterative-deepening search with respect to the plausibility and the locality of mode/type constraints.

With a few instances of a program, this framework could be used to infer a correct form of the program from a more ambiguous representation than an erroneous program (Section 6.3.1).

We embodied our framework in the Kima system, an automated error correction system for concurrent logic programs. Kima has three significant features:

- Kima corrects a few wrong occurrences of variable symbols in the absence of explicit mode/type declarations,

- Kima corrects both multiple independent errors in a single execution and k interdependent errors by depth-k search, and
- Kima can work on a fragment of a program.

We confirmed the effectiveness of our framework by experiments with Kima. By the exhaustive experiments for simple programs and experiments randomly chosen for rather large programs, we showed that

- Kima proposed one or a few alternatives to an erroneous program,
- heuristic rules we presented were quite effective not only in enhancing the quality of alternatives but also in optimizing the search of alternatives, and
- the density of (well-moded and well-typed) plausible programs in the syntactical neighborhood of the original (correct) program was very low.

We have released the Kima system as a freeware from
`http://www.ueda.info.waseda.ac.jp/~ajiro/study-e.html` .

6.1.1 Experiences of Implementing Kima in KL1

Kima is a KL1 program (of 5,200 lines long) that consists of 31 modules. KL1 has an efficient compiler named KLIC, KL1-to-C compiler system. The KLIC system has achieved high portability by employing C as an intermediate language, and can be worked on most popular Unix systems with gcc or some other C compilers.

KL1 and the KLIC system have provided the nice platform for describing the search problem (symbol processing) and for our experiments. Although KL1 has lost the feature of the automatic exhaustive search possessed inherently by Prolog, implementing Kima in KL1 might not be so inefficient in describing the search problem. This is because Kima requires detailed control of iterative-deepening A* search. In this case, even Prolog should explicitly deal with a certain part of the search procedure.

Kima has been implemented to correct KL1 programs by assuming strong moding and typing of Moded Flat GHC. KL1 is designed based on Flat GHC, and the debugging of KL1 programs turned out to benefit from moding and typing. In reality, Kima is nor well-moded nor well-typed

as a whole, but this is not an essential problem. Kima can deal with most features of KL1, and yet several built-in predicates cause their usage to be non-well-moded and non-well-typed. However, this problem could be solved in the following (rather ad hoc) way even for the current specification of KL1. We define, in a certain module, new well-moded and well-typed built-in predicates that have the same function as "ill-mannered" built-in predicates by using the ill-mannered built-in predicates themselves. We then prepare the (principal) mode constraints of the new predicates. In analyzing a program (set of modules), if we do not include the module of the new predicates but use the mode and type constraints imposed by the new predicates for their usage, the program can be well-moded and well-typed.

Kima's function of error correction is rather experimental, but the function of locating errors by computing minimal inconsistent subsets was very useful in developing Kima itself.

6.2 Related Work

6.2.1 Algorithmic and Declarative Debugging

Algorithmic debugging was first proposed by Shapiro [27] for diagnosing wrong and missing answers in Prolog. This technique locates the source of errors by performing the "binary search" of a proof tree in execution of a Prolog program. Basically, algorithmic debugging technique analyzes the difference between the intended and observed behaviors. Debugging with (partial or abstract) specification has been studied as declarative debugging for many programming languages. For instance, debugging of (concurrent or constraint) logic programs is found in [11, 19, 21].

Analysis of malfunctioning systems based on their intended logical specification has been studied in the field of artificial intelligence [22] 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 also to analyze the differences between intended and observed behaviors based on the specification of the system.

6.2.2 Debugging Type Errors

Type declaration can be thought of as a kind of partial specification. In languages equipped 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 [37, 4, 9, 14, 38]. 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. Such extension is necessary because Milner's type checking algorithm $\mathcal{W}$ [15] is not eligible to trace (backward) the deduction steps. The general polymorphism introduced by `let` constructor makes that problem difficult, because the polymorphic environment creates global dependencies in a type reconstruction tree.

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 (Section 3.4 and 4.3). Yang presented a manifesto of seven properties for a good type error reporting system to have [39]. Our system has all the properties as far as a minimal inconsistent subset (or the intersection of subsets as discussed in Section 5.1.4) is small.

A recent work by Chitil [7] defined a *compositional* type system in Haskell. His type system seems to be similar to our constraint-based mode/type system. His system managed, instead of types, *typing* relation between symbol occurrences and types for compositional (i.e., incremental) type reconstruction. His system solved the problem of `let`-polymorphism by copying the subtree created by a polymorphic (type) variable to every occurrence of the variable in a type reconstruction tree. This solution is very similar to the way of handling polymorphism in our framework. Further, he proposed to pinpoint the source of errors by applying classical algorithmic debugging to a type reconstruction tree.

A soft typing approach introduces static type systems to dynamically typed languages such as Lisp and Scheme [5, 1]. The MrSpidey system [10] has a programming environment that visually presents the explanation of type errors of Lisp programs based on soft typing and set-based analysis [12]. In this approach, debugging is performed interactively because the judgment 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 the flexibility 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 [28]. 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.

6.2.3 Automated Programming

Automated programming is a technique of programming by partial specification such as examples, demonstration, trace information, etc. At present, automated programming technique is not studied hard, but there is one field where the technique has achieved a success. That is the field of visual programming and (graphical) user interface. Programming by examples, demonstration, rehearsal has a great advantage in that field, because the textual representation of the GUI part in a program code is far from the actual GUI.

There are currently two techniques related to automated programming. They are Inductive Logic Programming (ILP) and Genetic Programming (GP). Their subjects are not a programming itself, but mode and type information is beneficial to optimization for both techniques [18, 16].

Inductive logic programming [17, 23] is the latest system for inductive inference that infers rules from examples. The ILP system succeeded to Model Inference System (MIS) by Shapiro [25, 26]. The subject of ILP is *conceptual learning*. That is, when given positive and negative examples, the ILP system computes the rules of classifying examples as a logic program both by generalizing the positive examples and by specializing the rules not to include the negative examples.

Our framework might be formalized as genetic programming [13]. Fitness values would be computed based on mode/type correctness and the plausibility criteria for our framework. When given parts of a program (i.e.,

geneses), the GP system *evolves* them into an optimum program under a certain fitness evaluation by using genetic manipulations such as mutation and crossing. However, the GP system is not for generic programming environment in the sense that whether to succeed in the evolution strongly depends on the choice of geneses.

6.2.4 Miscellaneous

Incidentally, we heard from a referee that some Fortran compilers of the 1970's were 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.

6.3 Future Work

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.

6.3.1 Automated Variable Placement

We expect that most variable occurrences in a predicate definition could automatically be specified by actually executing high-priority alternatives with an interpreter or a compiler. Suppose that a program is developed in a bottom up fashion and that the mode values of the arguments of predicates called from the predicate in question have already been solved correctly.

For example, suppose the instance of an `append` program is given as:

```
:- instance append(in([a,b,c]),in([d,e]),out([a,b,c,d,e])).
```

The function symbols "`in/1`" and "`out/1`" represent their arguments are the instances of input and output, respectively. Next, the `append` program with variable occurrences not specified (called *predicate schema*) is given as:

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

Here, the variable X means an unspecified variable. Then, the mode values of most paths are determined by the instance.

All paths whose mode values cannot be determined even with the instance are those in the arguments of built-in (or user-defined) polymorphic predicates such as unification body goals. However, the moding rules (HF), (BF) and (BU) (other than (HV), (GV) and (BV) as to variables) in Figure 2.2 may determine the mode values of paths that have not yet been determined by the instance. In the `append` case, the constraint below

$$m(\langle =_2, 1\rangle) = out$$

is derived from the two constraints:

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

Also, if unspecified variables Xs are at the same position on both sides of commutative predicates like $X =_1 X$, then, without the loss of generality, we can assume that the mode value on the left-hand side $\langle =_1, 1\rangle$ is *out* mode and the value on the right-hand side $\langle =_1, 2\rangle$ is *in* mode.

The number of different variables in a clause is at least the number of the paths of variable occurrences that are constrained to the mode value *out* (OUT) in a clause body. If, in a clause head, there are paths of variable occurrences constrained to the mode value *in* (IN), the number of different variables increases by at least one. Now, the predicate schema takes one step forward as:

```
R1: append([],      Y1,X) :- true | V1 =1 X.
R2: append([Y1|Y2],Y3,X) :- true |
        V1 =2 [Z1|Z2],append(X,X,V2).
```

Here, there are four kinds of variables, `V`n, Y_n, Z_n, and X for $n \geq 1$. The variable `V`n is specified, and the others are unspecified.

First, the variable `V`n means that the path of its occurrence has *out* mode in a clause body. The variable `V`n is a fixed, specified variable that is never

mutated (but may be mutated into "_" for penalty points to be improved). Second, the variable Y_n means that the path of its occurrence has *in* mode in a clause head. The variable Y_n is either promoted to the specified variable `Y`n or mutated into other Y_n in the same clause head. Third, the variable Z_n means that the mode value of the path of its occurrence has not been determined yet. The variable Z_n is either promoted to the specified variable `Z`n or mutated into other variables, `V`n, `Y`n or `Z`n in the same clause. Fourth, the variable X means that the path of the variable has either *out* mode in a clause head or *in* mode in a clause body. The variable X must be mutated into the other variables `V`n, `Y`n or `Z`n.

If we think of definition clauses as suspected clauses indicated by a minimal inconsistent subset (or a group of subsets), the search of a complete definition is the search of alternatives in the debugging sense. For each clause l_i in a predicate definition, we compute the set $\mathcal{L}^i_{h,d}$ of mutated clauses by the algorithm in Figure 4.6. There are two differences from the original way of mutating variable occurrences in the error correction. One is that the mutation is performed under the mutating rules above. The other is that the (partial) specification of modes and types has been provided by the instance. So, we redefine the consistent set $\mathcal{L}^i_{h,d}$ of mutated clauses, each of which is mode and type consistent with the instance (and is observing Detection Rules).

In the initial state of this search, different (singleton) variables should temporarily be placed on the occurrences of unspecified variables including X, because singleton variables impose a lot of penalty points on the initial state and help the A* search efficiently prune implausible mutations. The depth d of the search is the total number of unspecified variables, Y_n, Z_n, and X, in the clause. If we consider the mutation of a variable `V`n into "_", the depth is the total number of both unspecified and specified variables.

For definition clauses $\{l_1, \ldots, l_n\}$, we then define a set $\mathcal{Q}_i$ of mutated definition clauses of a predicate as:

$$\mathcal{Q}_i = \{l_1, \ldots, l_n | \; l_1 \in \mathcal{L}^1_{h_1,d_1}, \ldots, l_n \in \mathcal{L}^n_{h_n,d_n}, \; \sum_{j=1}^{n} h_j = n + i - 1\},$$

where d_i is the total number of unspecified variables in the clause l_i. In the `append` case, the set $\mathcal{L}^1_{1,3}$ consists of the single (consistent and intended) clause of R_1:

(1) `append([],Y1,V1) :- true | V1=`$_1$`Y1.`

The set $\mathcal{L}^2_{1,8}$ consists of the six consistent clauses of R_2:

(1) `append([Y1|Y2],Y3,V1):- true | V1=`$_2$`[Y1|V2],append(Y3,Y2,V2).`

(2) `append([Y1|Y2],Y3,V2):- true | V1=`$_2$`[Y1|Y3],append(V1,Y2,V2).`

(3) `append([Y1|Y2],Y3,V1):- true | V1=`$_2$`[Y1|V2],append(Y2,Y3,V2).`

(4) `append([Y1|Y2],Y3,V2):- true | V1=`$_2$`[Y1|Y3],append(Y2,V1,V2).`

(5) `append([Y1|Y2],Y3,V2):- true | V1=`$_2$`[Y1|Y2],append(V1,Y3,V2).`

(6) `append([Y1|Y2],Y3,V2):- true | V1=`$_2$`[Y1|Y2],append(Y3,V1,V2).`

With the integrated assistance of an interpreter or a compiler, the intended combination, (1) of R_1 and (3) of R_2, could be obtained by executing six (1×6) alternatives (i.e., six definitions) after the check of their well-modedness and well-typedness of the whole program.

Let clauses $\{l_1, \ldots, l_n\}$ be a predicate schema with specified and unspecified variables, and L_0 be all clauses but $\{l_1, \ldots, l_n\}$. Then, given the instance(s) and predicate schema, an algorithm for this automated variable placement is outlined in Figure 6.1.

6.3.2 Error Correction of Function Occurrences

The correction of function occurrences (including constants) might be realized with type information provided by the instances of a pair of input and output constraints. As far as the structure of the abstract syntax tree of an erroneous program is preserved, there are three kinds of the error correction of function occurrences, that is, the correction of

1. a function occurrence to a function occurrence,
2. a constant to a variable occurrence, and
3. a variable occurrence to a constant.

The second correction is possible even by the technique similar to the current version of Kima. In the first, the correction to function symbols except constants is not so difficult with type information. The main reason why the correction of numbers 1 and 3 is difficult is that we cannot choose an appropriate constant symbol to mutate into especially for integer numbers used in

Compute the mode values of the paths of variable occurrences with the instances and the moding rules (HF), (BF) and (BU);

Place specified variable, $\mathtt{V}n$, and unspecified variables, Y_n, Z_n and X based on the mode values computed above;

$h \leftarrow 1$;
while a solution is not found $\wedge\, h \leq MAX_h$ **do**
  Generate sets $\mathcal{L}^i_{h,d_i}$ for $1 \leq i \leq n$;

  Reset the sets $\mathcal{L}^i_{h,d_i}$ by clauses consistent with both the instances and predicates called from the predicate schema;

  Generate sets $\mathcal{Q}_1, \ldots, \mathcal{Q}_h$ of mutated definition clauses from the sets $\mathcal{L}^1_{h,d_1}, \ldots, \mathcal{L}^n_{h,d_n}$;

  for each $q \in \mathcal{Q}_h$ **do**
    if $mcs(L_0 \cup q)$ is consistent $\wedge$ $tcs(L_0 \cup q)$ is consistent **then**
      if the results of executing a program $(L_0 \cup q)$ matches the instances **then**
        Report the predicate definition q as a solution
      fi
    fi
  end for;
  $h \leftarrow h + 1$
end while

Figure 6.1: Algorithm for automated variable placement

most programs. However, in reality, the most occurrences of constants have charastristic values such as 0, 1, `[]`, `[[]]`, {}, etc. So, with the instances of a pair of input and output, the correction of function occurrences might be possible by executing alternatives.

Other kinds of error correction that cause the structure of the abstract syntax tree to mutate are again difficult. The elimination of symbol occurrences would relatively be easy, but supplying appropriate predicates is not because of the enormous search space of programs. It might be possible to supply a small number of function symbols, for example, mutating `X` into `[X]` rather ad hoc. To supply appropriate predicates and their arguments is the very topic of genetic programming and inductive inference.

6.3.3 Applicability to Other Languages

We take an interest in the applicability of our framework to other programming languages, especially tyepd functional languages such as ML. Although our framework could apparently be applied to the languages equipped with strong typing, there are many characteristic problems of each programming language to solve. The first subject of automated debugging is the target of debugging and the definition of simple errors, while the final goal is to keep the number of alternatives small (by additional heuristics if necessary). Furthermore, the alternatives have to be proposed in a certain short time.

In our framework, two techniques play important roles in automated debugging. One is to locate errors by computing minimal inconsistent subsets. This technique works on the ground that small number of suspected constraints can be obtained. The number of the elements of a minimal subset depends on the shape of a mode/type graph, which is wide but shallow. In contrast, the type graphs of functional programs are deep in general, and the "explanation" of errors is prone to be long.

The other is the heuristics used for the refinement of alternatives and for the optimization of searching alternatives. Intuitively, similar techniques seem to be necessary also for other languages. Automated error correction of even variables would be difficult without heuristics. Thus, we conclude that much consideration is necessary for applying our framework to other languages.

References

[1] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pages 167–173. ACM, 1994.

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

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

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

[5] R. Cartwright and M. Fagan. Soft Typing. In *Proc. ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation (PLDI'91)*, SIGPLAN Notice 26(6), pages 268–277. ACM, 1991.

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

[7] O. Chitil. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proc. Sixth Int. Conf. on Functional Programming (ICFP'01)*, pages 193–204. ACM Press, 2001.

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

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

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

[11] M. P.J. Fromherz. Towards Declarative Debugging of Concurrent Constraint Programs. In *Proc. First Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'93)*, number 749 in LNCS, pages 88–100. Springer, 1993.

[12] N. Heintze and J. Jaffar. Set Constraints and Set-Based Analysis. In *Proc. Principles and Practice of Constraint Programming, Second Int. Workshop (PPCP'94)*, number 874 in LNCS, pages 281–298. Springer, 1994.

[13] J. Koza. *Genetic Programming*. The MIT Press, Cambridge, MA, 1992.

[14] B. J. McAdam. Generalising Techniques for Type Debugging. In *First Scottish Functional Programming Workshop*, 1999. Also in P. Trinder, G. Michaelson, and H.-W. Loidl, editors, *Trends in Functional Programming*, pages 49–57, Intellect, 2000.

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

[16] D. J. Montana. Strongly Typed Genetic Programming. Technical Report 7866, Bolt Beranek and Newman, Inc., 1993.

[17] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.

[18] S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Methods. *J. of Logic Programming*, 19–20:629–679, 1994.

[19] L. Naish. A Declarative Debugging Scheme. *J. of Functional and Logic Programming*, 1997(3):1–27, 1997.

[20] G. D. Plotkin. A Structual Approach to Operational Semantics. DAIMI FN-19, Computer Science Dept., Aarhus Univ., Denmark, 1981.

[21] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-based Debugging of Constraint Logic Programs. In *Proc. Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer, 1999.

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

[23] C. Sammut, S. Hurst, D. Kedzier, and D. Michie. Learning Concepts by Asking Questions. In R. Michalski, J. Carbonnel, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Vol. 2*, pages 167–192. Morgan Kaufmann, 1986.

[24] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Conference Record of the Eighteenth Annual ACM Symp. on Principles of Programming Languages*, pages 333–352. ACM, 1991.

[25] E. Y. Shapiro. An algorithm that infers theories from facts. In *Proc. Seventh Int. Joint Conf. on Artificial Intelligence (IJCAI-81)*, pages 446–451. Morgan Kaufmann, 1981.

[26] E. Y. Shapiro. Inductive inference of theories from facts. Research Report 192, Yale University, 1981.

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

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

[29] K. Ueda. Designing a Concurrent Programming Language. In *Proc. Int. Conf. organized by the IPSJ to Commemorate the 30th Anniversary (InfoJapan'90)*, pages 87–94. Information Processing Society of Japan, 1990.

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

[31] K. Ueda. Experiences with Strong Moding in Concurrent Logic/ Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*, number 1068 in LNCS, pages 134–153. Springer, 1996.

[32] K. Ueda. Concurrent Logic/Constraint Programming: The Next 10 Years. In K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 53–71. Springer, 1999.

[33] K. Ueda. Resource-Passing Concurrent Programming. In *Proc. Fourth Int. Symp. on Theoretical Aspects of Computer Software*, number 2215 in LNCS, pages 95–126. Springer, 2001.

[34] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, 33(6):494–500, 1990.

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

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

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

[38] J. Yang and G. Michaelson. A visualisation of polymorphic type checking. *J. Functional Programming*, 10(1):57–75, 2000.

[39] J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved Type Error Reporting. In *Proc. 12th Int. Workshop on Implementation of Functional Languages*, pages 71–86. Aachner Informatik-Berichte, 2000.

Appendix A

Usage of Kima

A.1 Examples

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

Each alternative is separated by "-----". 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 checked for its value or instantiation 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*`)`

Kima always assume Detection Level 1, at which Detection Rule 1 is used without respect to command line options. The additional use of Detection Rule 2 (Detection Level 2) hinges on the options (see Section A.2).

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 ${}_nC_r$ 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 (Section A.2).

```
% 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, interdependent errors

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 performed, 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 $k \leq N$.

A.2 Details of Options

Below are the command line options available:

Option	Effect
`+mode`	use mode information only
`+type`	use type information only
`+dr`	use Detection Rule (singleton rule) only
`+mis`	present suspicious sources (Minimal Inconsistent Subsets)
`+d` N	search alternatives in up to depth N ($0 < N \leq 10$)
`+p` N	search alternatives with up to priority N (> 0)
`+pp` N	not search alternatives with penalty points more than N (≥ 0)

Any option other than the above is considered as a file name. If nonexistent file is given, the execution of Kima is interrupted. The option '`+dr`' involves the use of Detection Rule 2.

When no option is given, Kima assumes that options are as:

```
% kima +mode +type +dr +d 1 +p 1 +pp 20 FILE1 FILE2 ...
```

That is, depth-1 search of alternatives with the highest priority is performed by the information of modes, types and Detection Rules. If an alternative with penalty points 20 or less is not found, Kima terminates after reporting the failure of the search of alternatives.

The behavior of Kima including the default one is prescribed by the interpretation rules of command line options. All the rules are as follows:

- If neither the option '`+mode`', '`+type`' nor '`+dr`' is given explicitly, the three options are all considered to be given.
- If neither the option '`+mis`' nor '`+d`' is given, then the option '`+d 1`' is considered to be given. If you wants to obtain both minimal inconsistent subsets and alternatives at a time, the options should be given as '`+mis +d` N'.
- The default argument values of the options, '`+d`', '`+p`' and '`+pp`' are 1, 1 and 20, respectively.
- If the option, '`+d`', '`+p`' or '`+pp`' that requires its argument N is given without N, then the option itself is ignored.