In *Proc. Fourth International Conference on Principles and Practice of Constraint Programming (CP98)*, LNCS 1520, Springer-Verlag, Berlin, 1998, pp. 40–54.

Error-correcting Source Code

Yasuhiro Ajiro, Kazunori Ueda*, Kenta Cho**

Department of Information and Computer Science
Waseda University
4-1, Okubo 3-chome, Shinjuku-ku, Tokyo 169-8555, Japan
{ajiro,ueda}@ueda.info.waseda.ac.jp

Abstract. We study how constraint-based static analysis can be applied to the automated and systematic debugging of program errors.
Strongly moding and constraint-based mode analysis are turning to play fundamental roles in debugging concurrent logic/constraint programs as well as in establishing the consistency of communication protocols and in optimization. Mode analysis of Moded Flat GHC is a constraint satisfaction problem with many simple mode constraints, and can be solved efficiently by unification over feature graphs. We have proposed a simple and efficient technique which, given a non-well-moded program, diagnoses the "reasons" of inconsistency by finding minimal inconsistent subsets of mode constraints. Since each constraint keeps track of the symbol occurrence in the program that imposed the constraint, a minimal subset also tells possible sources of program errors. The technique is quite general and can be used with other constraint-based frameworks such as strong typing.
Based on the above idea, we study the possibility of *automated debugging in the absence of mode/type declarations*. The mode constraints are usually imposed redundantly, and the constraints that are considered correct can be used for correcting wrong symbol occurrences found by the diagnosis. As long as bugs are near-misses, the automated debugger can propose a rather small number of alternatives that include the intended program. Search space is kept small because constraints effectively prune many irrelevant alternatives. The paper demonstrates the technique by way of examples.

1 Introduction

This paper proposes a framework of automated debugging of program errors under static, constraint-based systems for program analysis, and shows how and why program errors can be fixed in the absence of programmers' declarations. The language we are particularly interested in is Moded Flat GHC [7][8] proposed in 1990. Moded Flat GHC is a concurrent logic (and consequently, a concurrent constraint) language with a constraint-based mode system designed by one of the authors, where modes prescribe the information flow that may be caused by the execution of a program.

* Supported in part by the Ministry of Education (grant No. 09245101) and Japan Information Processing Development Center (JIPDEC).

** Currently with Toshiba Corp.

Languages equipped with strong typing or strong moding[3] enable the detection of a type/mode errors by checking or reconstructing types or modes. The best-known framework for type reconstruction is the Hindley-Milner type system [3], which allows us to solve a set of type constraints obtained from program text efficiently as a unification problem.

Similarly, the mode system of Moded Flat GHC allows us to solve a set of mode constraints obtained from program text as a constraint satisfaction problem. Without mode declarations or other kinds of program specification given by programmers, mode reconstruction statically determines the read/write capabilities of variable occurrences and establishes the consistency of communication protocols between concurrent processes [8]. The constraint satisfaction problem can be solved mostly (though not entirely) as a unification problem over feature graphs (feature structures with cycles) and can be solved in almost linear time with respect to the size of the program [1]. As we will see later, types also can be reconstructed using a similar (and simpler) technique.

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

When a concurrent logic program contains bugs, it is very likely that mode constraints obtained from the erroneous symbol occurrences are incompatible with the other constraints. We have proposed an efficient algorithm that finds a minimal inconsistent subset of mode constraints from an inconsistent (multi)set of constraints [2]. A minimal inconsistent subset can be thought of as a minimal "explanation" of the reason of inconsistency. Furthermore, since each constraint keeps track of the symbol occurrence(s) in the program that imposed the constraint, a minimal subset tells possible sources (i.e., symbol occurrences) of program errors. Our technique can locate multiple bugs at once. The technique is quite general and can be used with other constraint-based frameworks such as strong typing.

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

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

[3] 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.

We can often take advantage of the redundancies and pinpoint a bug (Sect. 3) by assuming that redundant modes are correct. The next step worth trying is *automated error correction*. We can estimate the intended mode of a program from the parts of the program that are considered correct, and use it to fix small bugs, which is the main focus of this paper.

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

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

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

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

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

2 Strong Moding and Typing in Concurrent Logic Programming

We first outline the mode system of Moded Flat GHC. The readers are referred to [8] and [9] for details.

In concurrent logic programming, modes play a fundamental role in establishing the safety of a program in terms of the consistency of communication protocols. The mode system of Moded Flat GHC gives a polarity structure (that

determines the information flow of each part of data structures created during execution) to the arguments of predicates that determine the behavior of goals. A mode expresses this polarity structure, which is represented as a mapping from the set of *paths* to the two-valued codomain $\{in, out\}$. Paths here are strings of pairs, of the form $\langle symbol, arg \rangle$, of predicate/function symbols and argument positions, and are used to specify possible positions in data structures. Formally, the set P_{Term} of paths for terms and the set P_{Atom} of paths for atomic formulae are defined using disjoint union as:

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

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

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

A Flat GHC program is a set of clauses of the form h :- G | B, where h is an atomic formula and G and B are multisets of atomic formulae. Constraints imposed by a clause h :- G | B are summarized in Fig. 1. Rule (BU) numbers unification body goals because the mode system allows different body unification goals to have different modes. This is a special case of mode polymorphism that can be introduced into other predicates as well [2], but in this paper we will not consider general mode polymorphism because whether to have polymorphism is independent of the essence of this work.

For example, consider a quicksort program defined as follows:

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

From the entire definition, we obtain 53 constraints which are consistent. We could regard these constraints themselves as representing the principal mode of the program, but the principal mode can be represented more explicitly in terms of a mode graph (Fig. 2). Mode graphs are a kind of feature graphs [1] in which

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

(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

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

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

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

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

1. a path (in the graph-theoretic sense) represents a member of P_{Atom},
2. the node corresponding to a path p represents the value $m(p)$ ($\downarrow = in$, $\uparrow = out$),
3. each arc is labeled with the pair $\langle symbol, arg\rangle$ of a predicate/function symbol and an argument position, and may have a "negative sign" (denoted "•" in Fig. 2) that inverts the interpretation of the mode values of the paths beyond that arc, and
4. a binary constraint of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$ is represented by letting p_1 and p_2 lead to the same node.

Mode analysis proceeds by merging many simple mode graphs representing individual mode constraints. Thus its decidability is guaranteed by the decidability of the unification algorithm for feature graphs. The principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary (i.e., constraint on the mode value of, or the submode at, a particular path) or binary (i.e., constraint between the submodes at two particular paths). Space limitations do not allow us to explain further details, which can be found in [9].

A type system for concurrent logic programming can be introduced by classifying a set *Fun* of function symbols into mutually disjoint sets $F_1, \ldots, F_n$. A type

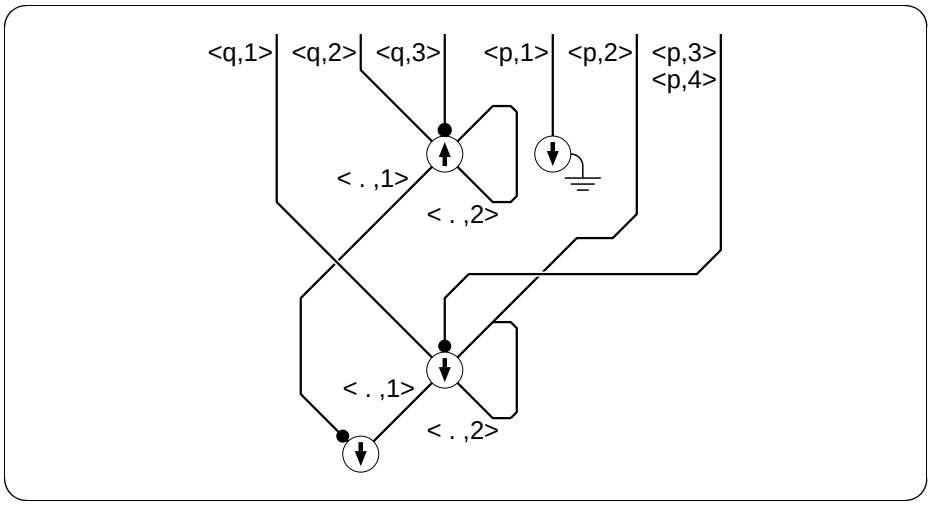

Fig. 2. The mode graph of a quicksort program. `q` stands for `qsort` and `p` stands for `part`. The mode information of the toplevel predicate and unification goals is omitted.

here is a function from P_{Atom} to the set $\{F_1, \ldots, F_n\}$. Like principal modes, principal types can be computed by unification over feature graphs. Constraints on a well-typing τ are summarized in Fig. 3. The choice of a family of sets $F_1, \ldots, F_n$ is somewhat arbitrary. This is why moding is more fundamental than typing in concurrent logic programming.

Mode and type analyses have been implemented as part of *klint*, a static analyzer for KL1 programs [11].

(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)	$\forall q \in P_{Term}\big(m(p'q) = in \Rightarrow \tau(pq) = \tau(p'q)\big)$, for a variable occurring both at p in h and at p' in G.
(BU_τ)	$\tau/\langle =_k, 1\rangle = \tau/\langle =_k, 2\rangle$, for a unification body goal $=_k$.

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

3 Identifying Program Errors

When a concurrent logic program contains an error, it is very likely (though not always the case) that its communication protocols become inconsistent and the set of its mode constraints becomes unsatisfiable. A wrong symbol occurring at some path is likely to impose a mode constraint inconsistent with correct constraints representing the intended specification.

A minimal inconsistent subset of mode constraints can be computed efficiently using a simple algorithm[4]. Let $C = \{c_1, \ldots, c_n\}$ be a multiset of constraints. Algorithm 1 below finds a single minimal inconsistent subset S from C when C is inconsistent. When C is consistent, the algorithm terminates with $S = \{\}$. $false$ is a self-inconsistent constraint used as a sentinel.

Algorithm 1:

$c_{n+1} \leftarrow false$;
$S \leftarrow \{\}$;
while S is consistent **do**
  $D \leftarrow S$; $i \leftarrow 0$;
  while D is consistent **do**
    $i \leftarrow i + 1$; $D \leftarrow D \cup \{c_i\}$
  end while;
  $S \leftarrow S \cup \{c_i\}$
end while;
if $i = n + 1$ **then** $S \leftarrow \{\}$

The readers are referred to [2] for a proof of the minimality of S, as well as various extensions of the algorithm. Note that the algorithm can be readily extended to finding multiple bugs at once. That is, once we have found a minimal subset covering a bug, we can reapply the algorithm to the rest of the constraints.

In the algorithm, the merging of constraint sets and the checking of their consistency are realized mostly as the unification of mode graphs and the checking of its success/failure. Although the algorithm is quite general, its efficiency hinges upon the fact that there is a pair of efficient algorithms for computing the union of constraint sets and checking its consistency.

Our experiment shows that the average size of minimal inconsistent subsets is less than 4, and we have not yet found a minimal inconsistent subset with more than 11 elements. The size of minimal subsets turns out to be independent of the total number of constraints, and most inconsistencies can be explained by constraints imposed by a small range of program text.

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

[4] The algorithm described here is a revised version of the one proposed in [2] and takes into account the case when C is consistent.

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

Algorithm 2 described below combines these two alternative policies of pinpointing. To reduce the amount of computation, we do not compute all minimal subsets; instead, for each element (say s_i) of the initial inconsistent subset S, we execute Algorithm 1 after removing s_i from C, which will lead to another minimal subset if it exists. Thus Algorithm 2 simultaneously computes constraints suspected by the two policies.

Let $S = \{s_1, \ldots, s_m\}$ be a minimal subset obtained by Algorithm 1, and `getminimal`(C) be a function which computes a minimal inconsistent subset from a multiset C of constraints using Algorithm 1 above:

Algorithm 2:

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

Here, T is a *multiset* of constraints what serves as counters of the numbers of constraints occurring in S and (various versions of) S', and $\dot{\cup}$ is a multiset union operator. T records how many times each constraint occurred in different minimal subsets. Under Policy 2, constraints with more occurrences in T are more likely to be related to the source of the error.

Algorithm 2 is useful in locating multiple bugs at once. That is, once we have obtained a minimal inconsistent subset S, we can apply Algorithm 2 to refine the subset and remove only those constraints in the refined subset from C.

When Policy 1 outputs a single constraint imposed by an erroneous symbol occurrence, we need not consider Policy 2. However, there are cases where Policy 1 outputs no constraints or more than one constraint, in which case Policy 2 may better tell which constraints to suspect first.

Algorithm 2 is not always able to refine the initial set S, however. For instance, when S is the only minimal inconsistent subset, the algorithm will output all the elements of S by Policy 1 and will find no alternative subset by Policy 2. Fortunately, this is not a serious problem because S is usually quite small.

4 Automated Debugging Using Mode Constraints

Constraints that are considered wrong can be corrected by

- replacing the symbol occurrences that imposed those constraints by other symbols, or
- when the suspected symbols are variables, by making them have more occurrences elsewhere (cf. Rule (BV) of Fig. 1).

In this paper, we focus on programs with a small number of errors in variables and constants; that is, we focus on errors in terminal symbols in abstract syntax trees. This may seem restrictive, but concurrent logic programs have quite flat syntactic structures (compared with other languages) and instead make heavy use of variables. Our experience tells that a majority of simple program errors arise from the erroneous use of variables, for which the support of a static mode system and debugging tools are invaluable.

An algorithm for automated correction is basically a search procedure whose initial state is the erroneous program, whose operations are the rewriting of the occurrences of variables or constants, and whose final states are well-moded programs[5]. This can be regarded also as a form of *abductive reasoning* which, from a presumably correct mode constraint B and the moding rules of the form "if A then B" (or "B for A") as shown in Fig. 1, infers a syntactic constraint A that is considered correct.

The symbols to be substituted in the correction are chosen from the constants or other variables occurring in the same clause. When the symbol to be rewritten occurs in the head, we should also consider replacement by a fresh variable. We don't have to try to form the mode graphs of all the alternative programs; from the set $C \setminus S$, we can derive a *replacement guideline*, namely simple constraints to be satisfied by the substituted symbol. Any replacement that violates the guideline will not lead to a well-moded program and can be rejected immediately.

Error correction may require the rewriting of more than one symbol occurrence. We perform iterative-deepening search with respect to the number of rewritings, because the assumption of near-misses implies that a simpler correction is more likely to be the intended one. These ideas have been partially implemented in the *kima* analyzer for KL1 programs [10].

5 Using Constraints Other Than Modes

When error correction requires the rewriting of more than one symbol occurrence, the iterative-deepening search may report a large number of alternative solutions, though they always include an intended one.

Using both the mode system and the type system reduces the number of alternatives greatly. Modes and types capture different aspects of a program, and rather few of well-formed programs are both well-moded and well-typed. We can expect that there are only a small number of well-moded and well-typed program syntactically in the 'neighborhood' of the given near-miss program.

[5] Here, we assume that errors can be corrected without changing the shape of the abstract syntax tree, though we could extend our technique and allow occurrences of terminal symbols to be simply added or deleted.

The reason why a type system alone is insufficient should become clear by considering programs that are simple in terms of types such as numerical programs. The mode system is sensitive to the number of occurrences of variables (rule (BV) in Fig. 1) and can detect many errors that cannot be found by type analysis. However, even when the programs are simple in terms of types, types can be useful for inferring what constant should replace the wrong symbol.

Other heuristics from our programming experiences can reinforce the framework as well:

1. A singleton variable occurring in a clause body is highly likely to be an error.
2. A solution containing a variable occurring more than once in a clause head is less likely to be an intended one.

These heuristics are not as *ad hoc* as it might look; indeed they can be replaced by a unified rule on *constraint strength*:

- A well-moded solution with weaker mode constraints is more likely to be an intended one.

A singleton variable occurring at p in a clause body imposes a constraint $m/p = OUT$, which is much stronger than $m(p) = out$. Similarly, a variable occurring more than once at p_1, p_2, ... in a clause head imposes a constraint $m/p_i = IN$.

We could use more surface-level heuristics such as the similarity of variable names, but this is outside the scope of this paper.

6 Experiments and Examples

We show some experimental results and discuss two examples of automated debugging. The examples we use are admittedly simple but that can be justified. First, we must anyway start with simple examples. Second, we have found that most inconsistencies can be explained by constraints imposed by a small range of program text, as we pointed out in Sect. 3. So we strongly expect that the total program size does not make much difference in the performance or the quality of automated debugging.

6.1 Experiments

We applied the proposed technique to programs with one mutation in variable occurrences. We systematically generated near-misses (each with one wrong occurrence of a variable) of three programs (there are many ways of inserting a bug) and examined how many of them became non-well-moded, whether automated correction reported an intended program, and how many alternatives were reported. Table 1 shows the results. In the table, the column "total cases" shows the numbers of cases examined, and the column "detected cases" shows how many cases lead to non-well-moded programs. For non-well-moded programs, we examined how many well-moded alternatives were proposed by the automated debugger by depth-1 search. In this experiment, we did not apply Algorithm 2 to refine a minimal inconsistent subset.

The programs we used are list concatenation (append), the generator of a Fibonacci sequence, and quicksort. We used the definitions of predicates only, that is, we did not use the constraints that might be imposed by the caller of these programs.

The row "mode only" indicates the results using mode constraints only, except that when correcting errors we regarded singleton variables in clause bodies as erroneous. In this experiment, minimal inconsistent subsets, when found, always included constraints imposed by the wrong symbol occurrence, and the original, intended programs were always included in the sets of the alternatives proposed by the algorithm.

Table 1. Single-error detection and correction

Program	Analysis	Total cases	Detected cases	Proposed alternatives 1	2	3	4	5	6	7	≥8
`append`	mode only	57	33	16	4	1	0	5	4	2	1
	new variable	13	11	7	0	0	1	1	2	0	0
	mode & type	57	44	19	3	2	5	1	3	0	0
`fibonacci`	mode only	84	43	28	7	0	0	0	2	3	3
	new variable	15	14	6	3	0	0	0	2	2	1
	mode & type	84	57	34	2	0	2	2	3	0	0
`quicksort`	mode only	245	148	84	33	2	3	1	8	7	10
	new variable	45	43	24	2	0	3	2	4	3	5
	mode & type	245	189	93	33	5	9	0	5	2	1

A bug due to a wrong variable occurrence often results from misspelling (say the confusion of `YS` and `Ys`), in which case the original variable is likely to be replaced by a variable not occurring elsewhere in the clause. The row "new variable" shows the statistics of this case, which tells most errors were detected by mode analysis.

The row "mode & type" shows the improvement obtained by using types as well. The column "detected cases" shows that some of the well-moded erroneous programs were newly detected as non-well-typed. Note that the experiments did not consider the automated correction of well-moded but non-well-typed programs. For `fibonacci` and `quicksort`, we assumed that integers and list constructors belonged to different types. For `append`, we employed a stronger notion of types and assumed that the type of the elements of a list could not be identical to the type of the list itself.

The results show that the use of types was effective in reducing the number of alternatives. More than half of non-well-moded near-misses were uniquely restored to the original program. Thus, programmers can benefit much from the support of constraint-based static analysis by writing programs in a well-moded and well-typed manner.

6.2 Example 1 — Append

As an example included in the above experiment, we discuss an `append` program with a single error. This example is simple and yet instructive.

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

Algorithm 1 computes the following minimal inconsistent subset of mode constraints:

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

This tells that we should suspect the variables `X` and `Y` in Clause R_2. The search first tries to rewrite one of the occurrences of these variables (iterative-deepening), and finds six well-moded alternatives:

(1) R_2 : `append([A|X],Y,Z0) :- true | Z0=`$_2$`[A|Z],append(X,Y,Z).`
(2) R_2 : `append([A|Y],X,Z0) :- true | Z0=`$_2$`[A|Z],append(X,Y,Z).`
(3) R_2 : `append([A|Y],Y,Z0) :- true | Z0=`$_2$`[A|Z],append(Y,Y,Z).`
(4) R_2 : `append([A|Y],Y,Z0) :- true | Z0=`$_2$`[A|Z],append(Z0,Y,Z).`
(5) R_2 : `append([A|Y],Y,Z0) :- true | Z0=`$_2$`[A|Z],append(A,Y,Z).`
(6) R_2 : `append([A|Y],Y,Z0) :- true | Z0=`$_2$`[A|Z],append(Z,Y,Z).`

Types do not help much in this example, though Alternative (5) can be eliminated by an implicit type assumption described in Sect. 6.1 that list constructors and the elements of the list cannot occupy the same path. Alternatives (3), (4), (5) and (6) are programs that cause reduction failure for most input data, and can be regarded as less plausible solutions because of the two occurrences of `Y` in the clause heads that impose stronger constraints than intended.

What are Alternatives (1) and (2)? Alternative (1) is the intended program, and Alternative (2) is a program that merges two input lists by taking their elements alternately. It's not '`append`', but is a quite meaningful program compared with the other alternatives!

In this example, Algorithm 2, if applied, will detect Constraint (b) as the unique result of Policy 1. This means that there must be some problems with the variable `X`, which in turn means that `X` must either be removed or occur more than once. Search of well-moded programs finds the same number of alternatives, but the search space is reduced because we do not have to consider the rewriting between `Y` and variables other than `X`.

6.3 Example 2 — Quicksort

Next, we consider a quicksort program with two errors.

1: R_1 : `quicksort(Xs,Ys) :- true | qsort(Xs,Ys,[]).`
2: R_2 : `qsort([], Ys0,Ys ) :- true | Ys=`$_1$`Ys0.`
3: R_3 : `qsort([X|Xs],Ys0,Ys3) :- true |`
4: `part(X,Xs,S,L),qsort(S,Ys0,Ys1),`
5: `Ys2=`$_2$`[X|Ys1],qsort(L,Ys2,Ys3).`
(the unification should have been `Ys1=`$_2$`[X|Ys2]`)

Algorithm 1 returns the following minimal inconsistent subset:

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

This subset is inconsistent because two inconsistent constraints can be derived from it:

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

It is worth noting that this example is rather difficult—the minimal subset is rather large and Algorithm 2 does not find an alternative minimal subset. That is, there is no redundancy of mode constraints in the formation of the difference list representing the result.

Thus we cannot infer the correct mode of the path $\langle \texttt{qsort}, 2\rangle$ and other paths, and automated debugging should consider both of the possibilities, $m(\langle \texttt{qsort}, 2\rangle) = in$ and $m(\langle \texttt{qsort}, 2\rangle) = out$.

We consider the correction of both constants and variables here. It turns out that all depth-1 corrections are non-well-moded. There are six depth-2 corrections that are well-moded:

(1) Line 1: `quicksort(Xs,Ys) :- true | qsort(Xs,Zs,Zs).`
(2) Line 1: `quicksort(Xs,Ys) :- true | qsort(Zs,Ys,Zs).`
(3) Line 1: `quicksort(Xs,Ys) :- true | qsort(Xs,`c`,Ys).`
(4) Line 1: `quicksort(Xs,Ys) :- true | qsort(`c`,Ys,Xs).`
(5) Line 5: `Ys2=`$_2$`[X|Ys2],qsort(L,Ys1,Ys3).`
(6) Line 5: `Ys1=`$_2$`[X|Ys2],qsort(L,Ys2,Ys3).`

Here, c is some constant.

Typing doesn't help much for this example. The assumption that integers and list constructors should not occupy the same path does not exclude any of the above alternatives.

However, usage information will help. Suppose we know that `quicksort` is used as $m(\langle \texttt{quicksort}, 1\rangle) = in$ and $m(\langle \texttt{quicksort}, 2\rangle) = out$. This excludes

Alternatives (1), (2) and (4). We can also exclude Alternative (5) by static occur-check (`Ys2` occurs on both sides of unification).

Of the remaining, Alternative (6) is the intended program that sorts items in ascending order. It is interesting to see that Alternative (3) is a program for sorting items in *descending* order by choosing '`[]`', the simplest element of the list type, as the constant c. This is not an intended program, but is a reasonable and approximately correct alternative which should not be rejected in the absence of program specification.

7 Related Work

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

Analysis of malfunctioning systems based on their intended logical specification has been studied in the field of artificial intelligence [4] and known as model-based diagnosis. Model-based diagnosis has similarities with our work in the ability of searching minimal explanations and multiple faults. However, the purpose of model-based diagnosis is to analyze the differences between intended and observed behaviors. Our mode system does *not* require that the intended behavior of a program be given as mode declarations, and still locates bugs quite well.

Wand proposed an algorithm for diagnosing non-well-typed functional programs [12]. His approach was to extend the unification algorithm for type reconstruction to record which symbol occurrence imposed which constraint. In contrast, our framework is built outside any underlying framework of constraint solving. We need not modify the constraint-solving algorithm but just call it. Besides its generality, our approach has an advantage that static analysis does not incur any overhead for well-moded/typed programs. Furthermore, the diagnosis guarantees the minimality of the explanation and often refines it further.

Comparison between Moded Flat GHC and other concurrent logic/constraint languages with some notions of moding can be found in [2].

8 Conclusions and Future Work

We studied how constraint-based static analysis could be applied to the automated and systematic debugging of program errors in the absence of mode/type declarations. We showed that, given a near-miss Moded Flat GHC program, our technique could in many cases report a unique solution or a small number of reasonable solutions that included the intended program.

If a programmer declares the mode and/or type of a program, that information can be used as constraints that are considered correct. In general, such constraints are useful in obtaining smaller minimal inconsistent subsets. However, our observation is that constraints implicitly imposed by the assumption of

well-modedness (and well-typedness) is strong enough for automatic debugging to be useful.

It is a subject of future work to extend our framework to the correction of non-terminal program symbols (i.e., function and predicate symbols), mainly in terms of search space. It is yet to see whether the proposed framework works well for other programming paradigms such as typed functional languages and procedural languages, but we would claim that the concurrent logic/constraint programming paradigm benefits enormously from static mode/type systems.

Acknowledgment

The authors are indebted to Norio Kato for his comments on an earlier version of this paper.

References

1. Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.
2. Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs, In *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96)*, The MIT Press, 1996, pp. 215–229.
3. Milner, R., A Theory of Type Polymorphism in Programming. *J. of Computer and System Sciences*, Vol. 17, No. 3 (1978), pp. 348–375.
4. Reiter, R., A Theory of Diagnosis from First Principles. *Artificial Intelligence*, Vol. 32 (1987), pp. 57–95.
5. Somogyi, Z., Henderson, F. and Conway, T., The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *J. Logic Programming*, Vol. 29, No. 1–3 (1996), pp. 17–64.
6. Ueda, K., I/O Mode Analysis in Concurrent Logic Programming. In *Proc. Int. Workshop on Theory and Practice of Parallel Programming*, LNCS 907, Springer, 1995, pp. 356–368.
7. Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming (ICLP'90)*, The MIT Press, 1990, pp. 3–17.
8. Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.
9. Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*, LNCS 1068, Springer, 1996, pp. 134–153.
10. Ueda, K. and Cho, K. *kima* — Analyzer of Ill-moded KL1 Programs. Available from `http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H8-index-E.html`, 1997.
11. Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H9-index-E.html`, 1998.
12. Wand, M., Finding the Source of Type Errors, In *Proc. 13th ACM Symp. on Principles of Programming Languages*, ACM, 1986, pp. 38–43.

This article was processed using the LaTeX macro package with LLNCS style