JOSAI MATHEMATICAL MONOGRAPHS
Vol. **2** *(2000) pp. 159 - 168*

Automatic Algorithm Stabilization System

Hiroshi SEKIGAWA*
NTT Communication Science Laboratories

Kiyoshi SHIRAYANAGI†
NTT Communication Science Laboratories

Abstract

We propose a method to automatically convert unstable programs in symbolic computation into stable programs based on the stabilization method proposed by Shirayanagi and Sweedler. We have implemented a prototype of an automatic algorithm stabilization system whose target symbolic computation program is Maple, in C language using lex and yacc, and have reported experimental results.

1 Introduction

In symbolic computation, it is dangerous to naively use an approximation or numerical approach. "Reasonably approximate results" cannot be obtained if we *simply* evaluate an original algorithm on approximate inputs. This is because, even if a sequence converges to a given input, the sequence of the outputs for the initial sequence does not necessarily converge to the true output. We will refer to algorithms that have such instability as *unstable algorithms.*

Shirayanagi proposed a method for stabilizing Buchberger's algorithm [9][10]. The method uses interval computation with "zero rewriting," which is the rule of rewriting an interval into zero if zero lies within the interval. The underlying ideas of this method were generalized by Shirayanagi and Sweedler as a theory of stabilizing algebraic algorithms [11].

However, until now, the conversion of unstable algorithms into stable ones is carried out manually. In this paper, we propose a method to carry out this process automatically.

In Section 2, we review the stabilization method. In Section 3, we describe an idea to automatically stabilize algorithms and show experiments. Finally, in Section 4, we describe some future directions.

*sekigawa@cslab.kecl.ntt.co.jp
†shirayan@cslab.kecl.ntt.co.jp

2 Stabilization Method Review

2.1 Method of Stabilizing Algorithms

In this section, we review the stabilization method. Our approach converts an unstable algorithm into a new algorithm. If the new algorithm runs utilizing increasingly accurate approximate computation, the output will converge to the exact output of the original algorithm. The stabilization method has three points.

1. The syntactic structure of the algorithm is unchanged.

2. The coefficients are converted to *interval-coefficients* in the data set.

3. *Rewriting* is performed prior to predicate evaluation.

Interval-coefficients are coefficients which have the form of intervals from interval analysis, see, for example [1]. Steps (1) and (2) coincide with existing interval methods and in general (1) and (2) alone are not sufficient for stabilization. The key is (3). This is a method which rewrites an interval to a new interval at the *discontinuous point(s)* of a predicate. The discontinuous points of a predicate, such as 0 in a conditional instruction "If $X = 0$ then ...", are points where the execution path of the algorithm may branch upon evaluation of the predicate. A common cause of algorithm instability is that approximate computation causes a predicate to be evaluated incorrectly and the algorithm runs on a wrong execution path. *Rewriting* moderates the effect of predicate discontinuity. It rewrites an interval-coefficient into (an interval signifying) the discontinuity point itself if the discontinuity point lies within the interval. Otherwise, rewriting leaves an interval-coefficient unchanged. This may have the same result as if exact computation with exact input had been done up until that point. In this case the modified algorithm passes through the branch-point in the same way as the original algorithm evaluated with exact computation on the exact input. For further details and more general theory see [11].

2.2 Manual Conversion of Programs

We utilize the stabilization method to stabilize algorithms. In real numbers or complex numbers, by transforming a predicate if necessary, we can assume that the discontinuous points of a predicate are empty or one point zero, and the only necessary rewriting is zero rewriting, that is, rewriting an interval into zero if zero lies within the interval. Since the syntactic structures of algorithms are unchanged, a slight modification of programs is enough for stabilization. We need to make functions that correspond to arithmetic operations and Maple's library functions that are inherently built in the system because Maple does not allow us to override the definitions of these operations and functions. We will explain the conversion procedure along with the following example.

Example 1

Consider the following Maple program:

```
example := proc(x)
local i;
    i := 0;
    while i < 1 do
        i := i + x;
    od;
    print(i);
end:
```

This program corresponds to the following procedure for given $x > 0$.

1. Set $i \leftarrow 0$.

2. While $i < 1$ set $i \leftarrow i + x$.

3. Print the value of i and terminate the procedure.

The instability occurs while testing the termination condition $i < 1$ in the while loop. Let x be $1/3$. When the value of i becomes 1, the while loop is terminated, the value 1 is printed, and the program is finished. If we use decimal floating-point approximation to $1/3$ with any high precision, the value of i when the while loop is terminated is approximate to $4/3$, rather than 1.

The resulting manual conversion is:

```
example := proc(x)
local i;
    i := 0;
    while bc_sign(i &- 1) < 0 do
        i := i &+ x;
    od;
    print(i);
end:
```

For representing intervals we use a type list, and for interval computations, we use an experimental interval arithmetic package "intpak" by Connell and Corless [3]. The symbols `&+` and `&-` stand for the addition and the subtraction for intervals, respectively. The function `bc_sign`, which returns the sign of an interval using zero rewriting, is:

```
bc_sign := proc(x)
    if x[1] > 0 then
        RETURN(1);
```

```
        elif x[2] < 0 then
            RETURN(-1);
        else            # x[1] <= 0 <= x[2]
            RETURN(0);  # zero rewriting
        fi;
    end:
```

Note that, as written in the comments, this program utilizes zero rewriting for deciding the sign of intervals.

3 Automatic Algorithm Stabilization

3.1 Idea

In this section, we will describe an idea for automatically stabilizing algorithms. Since the syntactical structure of algorithms is unchanged, we can easily do automatic conversion. The idea has two points:

- converting arithmetic operator names and Maple's library function names into function names that are previously prepared in the automatic stabilization system:
- preparing the sources of the above functions as a library of the automatic stabilization system.

Now, let us consider Example 1 in Section 2.2. The following conversion can be automatically and easily carried out.

```
example := proc(x)
local i;
    i := 0;
    while larger(1, i) do
        i := add2(i, x);
    od;
    print(i);
end:
```

Namely, expression `i < 1` is converted into `larger(1, i)` and addition `i + x` is converted into `add2(i, x)`.

Before explaining functions `larger` and `add2`, we explain a function that converts coefficients into intervals. For an input `x`, a type conversion function `convertstab` returns an interval with a tag, `[stab, [x1, x2]]`, where `x1` $\leq$ `x` $\leq$ `x2` with a specified precision. Here, we use tag `stab` because Maple has types but no type declarations.

Example 2

Consider the following fragment of a program:

```
⋮
if a = b then
    ...
fi;
⋮
```

Let `a` and `b` be lists, say, `[-0.1, 0.1]` and `[0, 0]`, respectively. If the type of `a` and `b` is just list, then we should judge `a` $\neq$ `b`; on the other hand, if `a` and `b` are intervals in the stabilization method, then we should judge `a = b` since the interval `a &- b` contains zero. To cope with this problem, we use tag `stab`, and convert the program as follows:

```
⋮
if equal(a, b) then
    ...
fi;
⋮
```

Here, the function `equal` looks like:

```
equal := proc(x, y)
    if (at least one of x and y is an interval with tag stab) then
        (after converting x or y into an interval with tag stab if necessary)
        RETURN(equalstab(x[2], y[2]));
        # x = [stab,[x1, x2]], y = [stab, [y1, y2]].
    else
        RETURN(evalb(x = y));
    fi;
end:
```

The function `equalstab` is:

```
equalstab := proc(x, y)
    if x[2] < y[1] then
        RETURN(false);
    elif x[1] > y[2] then
        RETURN(false);
    else
        RETURN(true); # zero rewriting
    fi;
end:
```

This program utilizes zero rewriting; see the following equation:

$$[x_1, x_2] - [y_1, y_2] = [x_1 - y_2, x_2 - y_1].$$

Namely, `x[2] < y[1]` and `x[1] > y[2]` means that the interval $x - y$ is completely in the negative region in the real line, and completely in the positive region in the real line, respectively; on the other hand, when both of the inequalities `x[2] >= y[1]` and `x[1] <= y[2]` are satisfied, we judge that the interval $x - y$ is zero because $x - y$ contains zero.

Modified programs are ready for both exact inputs and interval inputs in this method. Furthermore, if we use different tags for the same type, we can treat different rules of computations; for example, we can use a tag `strict` for intervals and when deciding the sign of an interval, if the interval contains zero, then we terminate the computation with an error message "cannot decide sign".

Next, we explain other initially prepared functions as a library in the automatic stabilization system. For example, function `larger` looks like:

```
larger := proc(x, y)
    if (x and y are numbers (not intervals with tag stab)) then
        RETURN(evalb(x > y));
    else
        (after converting x or y into an interval with tag stab if necessary)
        RETURN(largerstab(x[2], y[2]));
        # x = [stab,[x1, x2]], y = [stab, [y1, y2]].
    fi;
end:
```

In this case, we suppose that the types of inputs for this function are ordinal numbers or intervals with tag `stab`. Note that the type testing order is different from `equal`, since `x` and `y` are not numbers or intervals in general in `equal` case; they may be sets, lists, or all other types in Maple. The function `evalb` evaluates an expression as a Boolean expression, that is, evaluates `x > y` as true or false.

The function `largerstab` is:

```
largerstab := proc(x, y)
    if x[1] > y[2] then
        RETURN(true);
    else
        RETURN(false);
    fi;
end:
```

This program also utilizes zero rewriting. When the inequality `x[1] <= y[2]` is satisfied, that is, (1) when the interval $x - y$ is completely in the negative region in the real line, or

(2) when the interval $x - y$ contains zero, we judge that the interval x is not larger than the interval y; in the case of (2), we judge that $x - y$ is zero and x is not larger than y.

The function `add2` looks like:

```
add2 := proc(x, y)
    if (x and y are numbers (not intervals with tag stab)) then
        RETURN(x + y);
    else
        (after converting x or y into an interval with tag stab if necessary)
        RETURN(add2stab(x[2], y[2]));
        # x = [stab,[x1, x2]], y = [stab, [y1, y2]].
    fi;
end:
```

If we use "intpak" for interval computations, then `add2stab`, the addition between two intervals, is:

```
add2stab := proc(x, y)
    RETURN(x &+ y);
end:
```

Other arithmetic operations are also prepared in a similar way.

3.2 Subtle and Complicated Examples

In this section, we describe some subtle and complicated points. First, we consider the symbol "=". Note that we should not convert the symbol "=" into `equal` in some cases. Let us consider the following examples.

Example 3
Consider the function `subs`*, which substitutes subexpressions into an expression. The following is a Maple session:*

```
> subs(x = 1, sin(x) + x^2);

                    sin(1) + 1
```

We should not convert `x = 1` into `equal(x, 1)` in this case. Similar situations occur for functions, for example, `seq` and `subsop`.

A complicated example is a treatment of loops.

Example 4
Consider the following loop:

```
for i from a by s to b while c(i) do
    insideloop(i);
od;
```

This program corresponds to the following procedure:

1. Set $i \leftarrow a$.
2. If $i > b$ (when $s > 0$) or $i < b$ (when $s < 0$) or $c(i)$ is false, then terminate the procedure.
3. Do insideloop(i).
4. Set $i \leftarrow i + s$. Goto Step 2.

Note that the direction of the inequality in the loop termination condition changes according to the sign of s. The result of conversion is:

```
i := a;
while c(i) do
    if larger(mul(sig(s), i), mul(sig(s), b)) then
        break;
    fi;
    insideloop(i);
    i := add2(i, s);
od;
```

The function `mul` is a multiplication for two arguments, and we use the function name `sig` because `sign` is already used in Maple.

3.3 Implementation

We have implemented a prototype in C language on an HP9000/735, using lex as a lexical analyzer generator and yacc as a parser generator. For details of these tools see UNIX manuals or [2]. The Backus Naur form grammar of the Maple language is in [4]. Since the prototype uses only standard tools on UNIX, practically the same programs for the stabilization system can be compiled on other platforms; we have checked a DEC Alpha station and a Toshiba TECRA530 (Linux).

3.4 Experiments

We compare the quality of programs that are automatically and manually converted. The original program constructs two dimensional convex hulls using Graham's algorithm [6] (or see, for example, [8]).

Table 1: Cpu times of automatically/manually converted programs

number of points	100	1000	10000
automatic converted program (sec.)	10.4	128.2	3336.0
manual converted program (sec.)	9.1	110.8	2961.6
ratio	1.14	1.16	1.13

We have implemented Graham's algorithm in Maple V Release 3 on an HP9000/735. The size of the program is approximately 450 lines. It takes less than 0.1 seconds to automatically convert the program into a stabilized program. We show the cpu times for both of the automatically and manually converted programs for constructing convex hulls of 100, 1000 and 10000 points in Table 1. We use decimal floating-points with precision digits 10 for representing intervals and exclude the process times in converting inputs into intervals. The table shows the cpu times increase by approximately 15% with automatic conversion. The computation expense of type testing causes the increase in the cpu times.

4 Conclusion

We have implemented a prototypical automatic stabilization system and carried out some experiments.

Some future directions are:

- To carry out experiments for other algorithms than Graham's to examine the quality of the system.
- To enrich the library.
- To make the system user-friendly. For example, in the present prototype, users should decide whether a function should be converted in a program.
- To construct systems for other target symbolic computation systems, for example, Risa/Asir [7].
- To construct a preprocessor for the Gauss system [5] in Maple using the idea of the present paper.

References

[1] Alefeld, G. and Herzberger, J.: *Introduction to Interval Computations*, Academic Press, 1983.

[2] Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] Connell, A. E. and Corless, R. M.: An experimental interval arithmetic package in Maple, *Interval Computations*, No. 2, 1993, 120–134.

[4] Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B. L., Monagan, M. B. and Watt, S. M.: *Maple V Language Reference Manual*, Springer-Verlag, 1991.

[5] Grunts, D. and Monagan, M.: Introduction to Gauss, *SIGSAM Bulletin*, **28**, No. 2, 1994, 3–19.

[6] Graham, R. L.: An efficient algorithm for determining the convex hull of a finite planar set, *Inform. Proc. Letters*, **1**, 1972, 132–133.

[7] Noro, M. and Takeshima, T.: Risa/Asir—A computer algebra system, *Proc. ISSAC'92*, 1992, 387–396.

[8] Preparata, F. P. and Shamos, M. I.: *Computational Geometry*, Springer-Verlag, 1985.

[9] Shirayanagi, K.: An algorithm to compute floating point Gröbner bases, *Mathematical Computation with Maple V: Ideas and Applications* (Lee, T. ed.), Birkhäuser, 1993, 95–106.

[10] Shirayanagi, K.: Floating point Gröbner bases, *Mathematics and Computers in Simulation*, **42**, 1996, 509–528.

[11] Shirayanagi, K. and Sweedler, M.: A theory of stabilizing algebraic algorithms, *Technical Report 95-28, Mathematical Sciences Institute, Cornell University*, 1995.