Axiomatics of Exceptions

Wei Chen, United States of America

Abstract. This paper presents axiomatic proof rules for exception handling and argues that exceptions are no more than forward goto statements. The rules are defined within Hoare's native proof system with Dijkstra's weakest precondition model as a mathematical foundation.

Keywords: exception handling, exception proof rules, goto axioms, weakest precondition.

1 Introduction

We have made some crucial progress toward axiomatizing goto and its family of jump statements [1][2][3]. In this paper we present a very simple exception rule based on Clint and Hoare's mostly ignored goto rule presented in the main text of [4]. Exceptions are no more than so called *forward* jumps, i.e. where, roughly speaking, goto is encountered before its label. For forward jumps, termination is guaranteed, and no variant functions are needed. Importantly, forward jumps are compositional in that we can repeatedly apply the same single one-label rule. It is well known that the goto statement is largely used for exception handling in our common practice. We follow this path and lay down a solid mathematical foundation first on the goto statement. It is our great fortune that we have Dijkstra's weakest precondition model [6][7] handily available. The goto statement and other jump statements were skipped when the weakest precondition model was first conceived around 70's of last century by Dijkstra. However, the real power of the model lies actually in its ability to formally treat the goto statement, in our opinion, since we found it out that goto possesses more mathematical properties than all other programming constructs aggregated.

In the next section, we will postulate the weakest precondition and present some properties of the goto statement. We give methods on how to obtain goto's wp. Converting the weakest precondition of a goto statement to a Hoare triple yields an axiom of goto statements in Hoare's native proof system. In section 3, we discuss proof rules of goto statements especially for forward jumps in a *miraculous* style. Section 4 presents exception rules based on Clint and Hoare's originally devised intention. We show why exceptions are just like forward jumps. We also define the semantics for try-raise-catch statements found in modern programming languages. Following section 4, we show how to use exception rules in a case study. Section 6 sees some concluding remarks being drawn. We see little in publication of the subject but will make a comparison of approaches taken with [8] in the last section.

2 Mathematics beneath goto Statements

This section provides some background information which are already reported somewhere else [1-3] but useful in understanding the rest of the paper. Given a postcondition, a goto execution transfers the control to where its label is rather than the end of the goto statement, so the weakest precondition of a goto statement is the weakest precondition given at its label. In [3], the weakest precondition of the goto statement is defined by

$\text{wp(goto L, R)} \triangleq \text{wpL}$ **(goto rule)**

where wpL represents the weakest precondition at label L, i.e. wp(L:S, Q) for some computed Q to that point. Such a Q exists as we are dealing with total functions. This is much alike to any other rule of wp. For example, given S1;S2, whose wp is wp(S1, wp(S2, R)), as S1 or S2 changes, the entire wp could change. We also need yet determine the mid-point wp, say, wpMid, which is just wp(S2, R). The only difference is that we do not feel like writing down S in the goto rule. Whether goto reaches its end does not matter much since we are seeking a precondition. We can still claim that goto reaches the end but through a *miracle*.

We take a few more observations on this definition. First, the rule does not require the uniqueness for locations where wpL holds within a program, so theoretically it allows the same label to appear in multiple locations as long as the weakest precondition at each location is the same wpL. The goto statement can jump to any of such locations. So, L:L:S must be allowed in the language, among others. Such a forking jump, although we don't need it in this paper, may shed more light on non-determinacy and parallel composition. Our second observation is that the rule does not impose any scoping nor require where the label should be. So, it could be inside an alternation or repetition construct. For example, consider

$\text{prog1} \triangleq \text{do } x < 0 \rightarrow \text{L: } x := x+1 \text{ od; if } x > 0 \text{ then } x := -x\text{; goto L fi}$

which has a jump into the loop body. We calculate,

wp(prog1, post)
= "plug in prog1"
$\text{wp(do } x < 0 \rightarrow \text{ L: } x := x+1 \text{ od; if } x > 0 \text{ then } x := -x\text{; goto L fi, post)}$
= "sequential composition and if-then rules [(1)]"
$\text{wp(do } x < 0 \rightarrow \text{L: } x := x+1 \text{ od, } x>0 \wedge \text{wp(}x := -x\text{; goto L, post)} \vee 0\geq x \wedge \text{post)}$
= "sequential composition, goto, assignment rules"
$\text{wp(do } x < 0 \rightarrow \text{do L: } x := x+1 \text{ od, } x>0 \wedge \text{wpL}(x \leftarrow -x) \vee 0\geq x \wedge \text{post)}$
= "repetition rule[(1)]"
the strongest solution of Z: $[\ Z \equiv x < 0 \wedge \text{wp(L: } x := x+1, Z)$
$\vee\ x > 0 \wedge \text{wpL}(x \leftarrow -x) \vee x=0 \wedge \text{post}\ ]$
= "assignment rule, found wpL and plug in, which is $Z(x \leftarrow x+1)$"
the strongest solution of Z: $[\ Z \equiv x < 0 \wedge Z(x \leftarrow x+1)$
$\vee\ x > 0 \wedge Z(x \leftarrow x+1)\ (x \leftarrow -x) \vee x=0 \wedge \text{post}]$
= "substitution"

(1) The if-then rule: $\text{wp(if B then S fi, post)} \triangleq [\ B \wedge \text{wp(S, post)} \vee \neg B \wedge \text{post}\ ]$.
The repetition rule: $\text{wp(do B} \rightarrow \text{S od, post)} \triangleq$ the strongest solution X: $[X \equiv B \wedge \text{wp(S, X)} \vee \neg B \wedge \text{post}]$

the strongest solution of Z: [$Z \equiv x < 0 \wedge Z(x \leftarrow x+1)$
$\vee\ x > 0 \wedge Z(x \leftarrow -x+1) \vee x=0 \wedge \text{post}$]
= "solve the above equation by approximation"
post($x\leftarrow 0$)

Therefore, wp(prog1, post) = post($x\leftarrow 0$). Indeed, prog1 operationally behaves like an assignment of setting x to 0. Thirdly, goto does not satisfy Dijkstra's Law of Excluded Miracle, which stipulates that [wp(S, F) ≡ F] for any executable statement S where F denotes the universally false predicate. Any statement is called *miraculous* if it does not meet Law of Excluded Miracle. Miraculous statements are in general not executable, but goto, actually, is one of a few executable miracles, because the program containing goto does follow Law of Excluded Miracle. To show this, we need to define an auxiliary statement called miracle Q, which has the following wp rule.

wp(miracle Q, R) ≜ Q.

Then we have goto L = miracle wpL, and abort = miracle F. Now suppose our program looks like

SLL3 ≜ S0; L1:S1; L2:S2; L3:S3

with three labels. It is pretty easy to see, we have

wp(SLL3, post) = wp(S0, wpL1),
wpL1 = wp(S1, wpL2),
wpL2 = wp(S2, wpL3) and
wpL3 = wp(S3, post).

We use miracle to replace goto to obtain,

wp(SLL3, post) = wp(S0(goto L1←miracle wpL1, goto L2←miracle wpL2,
goto L3←miracle wpL3), wpL1)
wpL1 = wp(S1(goto L1←miracle wpL1, goto L2←miracle wpL2,
goto L3←miracle wpL3), wpL2)
wpL2 = wp(S2(goto L1←miracle wpL1, goto L2←miracle wpL2,
goto L3←miracle wpL3), wpL3)
wpL3 = wp(S3(goto L1←miracle wpL1, goto L2←miracle wpL2,
goto L3←miracle wpL3), post)

so (wpL1, wpL2, wpL3) is a solution of the following ternary recursive equations.

(X1, X2, X3) :
(X1 = wp(S1(goto L1←miracle X1, goto L2←miracle X2,
goto L3←miracle X3), X2)
X2 = wp(S2(goto L1←miracle X1, goto L2←miracle X2,
goto L3←miracle X3), X3)
X3 = wp(S3(goto L1←miracle X1, goto L2←miracle X2,
goto L3←miracle X3), post))

As usual, we designate the strongest solution(2) of X1, X2 and X3 under position-wise implications to be wpL1, wpL2 and wpL3, respectively. It is not too hard to see (F, F, F) is also a solution when post assumes F because miracle F = abort and S1, S2

(2) This is because our execution of a recursive program leads to such a strongest solution. As the Cartesian product of predicates forms a complete lattice under point-wise implication, the least fixed point, i.e. the so-called strongest solution, exists according to Knaster-Tarski fixed point theorem.

and S3 hypothetically preserve Law of Excluded Miracle. But no other solution could be stronger. Thus, wpL1 = wpL2 = wpL3 = F. We then have

wp(SLL3, F)
= “ above and rewriting ”
wp(S0(goto L1←miracle F, goto L2←miracle F, goto L3←miracle F), F)
= “ miracle F = abort ”
wp(S0(goto L1←abort, goto L2←abort, goto L3←abort), F)
= “ S0 hypothetically preserves Law of Excluded Miracle ”
F

It is straightforward to generalize this into the n-label case SLL ≜ $S_0;L_1:S_1;\dots L_n:S_n$. Since SLL is a canonical form and any program can be converted to its appearance, we have established the following theorem.

Theorem of Jump Executability

All goto statements are executable.

□

This theorem reflects the fact that the goto statement can indeed be implemented on every computing machinery. The unfortunate coining of Law of Excluded Miracle by Dijkstra probably contributes to such late recognition that goto is both miraculous and executable. The interested reader can refer to [3] for more details.

Fourthly, the goto axiom does exist for Hoare’s native proof system, on the contrary to popular and massive skepticism, and such axioms can easily be converted from its weakest precondition. We have

$$\frac{\text{< empty >}}{\{\ wpL\ \}\ \text{goto}\ L\ \{\ F\ \}}$$

for total correctness. For partial correctness

$$\frac{\text{< empty >}}{\{\ wlpL\}\ \text{goto}\ L\ \{\ F\ \}}$$

where wlpL is the weakest liberal precondition at the label L. Now we should become aware why { T } goto L { F } cannot be a goto axiom, first noticed in [10]. Executing a goto does require a precondition. It is not due to Hoare’s proof system but our own failure not to uncover those axioms earlier.

3 Proof Rules for goto Statements

Since we intend to use goto to describe exceptions, we first give some proof rules of goto. Clint and Hoare’s rule in an appendix of [4] states

$$\frac{\begin{array}{c}\{\ Q\ \}\ \text{goto}\ L\ \{\ F\ \} \vdash \{\ P\ \}\ S1\ \{\ Q\ \}\\ \{\ Q\ \}\ \text{goto}\ L\ \{\ F\ \} \vdash \{\ Q\ \}\ S2\ \{\ R\ \}\end{array}}{\{\ P\ \}\ S1;\ L{:}S2\ \{\ R\ \}}$$

Using the miracle statement, we can do some wonders. Noting that goto L = miracle wpL, we have

$$\frac{\begin{array}{c}\{\ Q\ \}\ \text{miracle wpL}\ \{\ F\ \} \vdash \{\ P\ \}\ S1(\text{goto}\ L \leftarrow \text{miracle wpL})\ \{\ Q\ \}\\ \{\ Q\ \}\ \text{miracle wpL}\ \{\ F\ \} \vdash \{\ Q\ \}\ S2(\text{goto}\ L \leftarrow \text{miracle wpL})\ \{\ R\ \}\end{array}}{\{\ P\ \}\ S1;\ L{:}S2;\{\ R\ \}}$$

Furthermore, we can and do choose Q to be wpL, then we have

$$\begin{array}{c}\{\ P\ \}\ \text{S1(goto L} \leftarrow \text{miracle wpL)}\ \{\ \text{wpL}\ \} \\ \{\ \text{wpL}\ \}\ \text{S2(goto L} \leftarrow \text{miracle wpL)}\ \{\ R\ \} \\ \hline \{\ P\ \}\ \text{S1; L:S2}\ \{\ R\ \}\end{array}$$

since LHS of the sleeping tee sign ⊢ is just an instance of the miracle axiom. Now that wpL becomes a new unknown, we can replace it back with Q again; thus, we have

$$\begin{array}{c}\{\ P\ \}\ \ \text{S1(goto L} \leftarrow \text{miracle Q)}\ \{\ Q\ \} \\ \{\ Q\ \}\ \text{S2(goto L} \leftarrow \text{miracle Q)}\ \{\ R\ \} \\ \hline \{\ P\ \}\ \text{S1; L:S2}\ \{\ R\ \}\end{array}$$

This is actually a rewriting of Clint and Hoare's goto rule for partial correctness [2]. When we take S2 as skip in the above, we obtain,

$$\begin{array}{c}\{\ P\ \}\ \text{S(goto L} \leftarrow \text{miracle R)}\ \{\ R\ \} \\ \hline \{\ P\ \}\ \text{S; L:skip}\ \{\ R\ \}\end{array}$$

We call every goto L contained in S a forward jump in that goto L gets executed first before its label appears. This rule is essentially a labeling rule, comparing its antecedent with consequent. Since forward jumps have no termination issue, this rule also characterizes total correctness for forward jumps. We will call this rule the forward SLL rule. Since we do not use the backward SLL rule in this paper, we will sometimes refer to this rule simply as the SLL rule.

4 Exception Proof Rules

The forward jump is used quite often in handling exceptions. The main rule by Clint and Hoare in [4], which capture the exception semantics, states

$$\begin{array}{c}\{\ P1\ \}\ \text{S1}\ \{R1\} \\ \{\ P1\ \}\ \text{goto L}\ \{\ F\ \} \vdash \{\ P\ \}\ \text{S2(goto L)}\ \{\ R2\ \} \\ \hline \{\ P\ \}\ \text{L:S1; S2(goto L);}\ \{\ R1 \vee R2\ \}\end{array}$$

Clint and Hoare's vision is presumably that S1 is taken as an exception block and contains no more goto L, and that execution starts from S2 and ends normally when S2 ends, but exceptionally ends when S1 ends after goto L is encountered. Although goto L appears like a backward jump, it is actually a forward jump in that S1 contains no goto L. So, we can re-write the consequent statement portion equivalently as

E_block ≜ S_block(goto L); goto Next; L:S_handler; Next:skip

We have two labels L and Next. For a postcondition R, we have wp at the label Next as R. So, by reverse applying the forward SLL rule, our proof obligation becomes for a precondition P

{ P } S_block(goto L); miracle R; L:S_handler { R }

In order to prove

{ P } S_block(goto L); miracle R; L:skip { Q }

we reverse apply the SLL rule, again. Our proof obligation is

{ P } S_block(goto L ← miracle Q); miracle R { Q }

Obviously, wpL is wp(S_handler, R). By the sequential composition rule, Q needs to be weaker than wpL; thus we have one more obligation to prove

$\{ Q \}$ S_handler $\{ R \}$

Finally, by the miracle rule, the first obligation can be written as

$\{ P \}$ S_block(goto L $\leftarrow$ miracle Q) $\{ R \}$

Putting pieces together, we obtain

$$\frac{\begin{array}{c}\{ P \}\ \text{S\_block(goto L} \leftarrow \text{miracle Q)}\ \{ R \} \\ \{ Q \}\ \text{S\_handler}\ \{ R \}\end{array}}{\{ P \}\ \text{E\_block}\ \{ R \}}$$

Modern languages use a try-raise-catch mechanism, which can be viewed as a variety of the above rule. For example,

try S_block(**raise** Exception(anException))

catch Exception(anExceptionParam)

    S_handler

yrt

can be modeled as

```
TryRaiseCatch  ≜  S_block(raise Exception(anException)
                                   ← ex ≔ anException; goto L);
                  goto Next;
               L: anExceptionParam ≔ ex; S_handler;
            Next: skip
```

Then we can define its semantics as the following proof rule by applying the SLL rule, plus other rules,

$$\frac{\begin{array}{l}[\ Q1 \Rightarrow Q2(\text{anExceptionParam} \leftarrow \text{ex})\ ] \\ \{ P \}\ \text{S\_block(raise anException} \leftarrow \text{miracle (Q1(ex} \leftarrow \text{anException)))}\ \{ R \} \\ \{ Q2 \}\ \text{S\_handler}\ \{R\}\end{array}}{\{ P \}\ \text{TryRaiseCatch}\ \{ R \}}$$

We can simplify it further to

$$\frac{\begin{array}{l}[\ Q1 \Rightarrow Q2\ (\text{anExceptionParam} \leftarrow \text{anException})\ ] \\ \{ P \}\ \text{S\_block(raise anException} \leftarrow \text{miracle Q1)}\ \{ R \} \\ \{ Q2 \}\ \text{S\_handler}\ \{R\}\end{array}}{\{ P \}\ \text{TryRaiseCatch}\ \{ R \}}$$

If we don't have parameter passing during raise, the above rule is just like that of E_block.

5 Case Study: Table Search

We are given a two-dimensional array A to find whether or not a given constant K exists in A. First, we formalize the problem specification in a Hoare triple $\{ x=A \}$ TSP $\{$

$x=A \wedge (found \equiv (\exists\, i, j :: A.i.j = K))$ }. Using the standard technique, we could have the following for TSP

```
{ $x = A$ }
$m := 0$; $found := false$;
{ inv: $(\forall\, i, j : 0 \le i < m \wedge 0 \le j < N : A.i.j \ne K) \wedge 0 \le m \le M \wedge \neg found$
  vf: $M-m$ }
do $m \ne M \rightarrow$ $n := 0$;
    { inv: $(\forall\, i, j : 0 \le i < m \wedge 0 \le j < n : A.i.j \ne K) \wedge 0 \le m \le M \wedge 0 \le n \le N$
                                                    $\wedge \neg found$
      vf: $N-n$ }
    do $n \ne N \rightarrow$
        " compare A.m.n with K: "
        if $x.m.n = K \rightarrow found := true$
        [] $x.m.n \ne K \rightarrow$ skip
           fi; $n := n + 1$
    od; $m := m + 1$
od { $m = M \wedge (\forall\, i, j : 0 \le i < m \wedge 0 \le j < N : A.i.j \ne K) \wedge 0 \le m \le M \wedge \neg found$ }
{ $x=A \wedge (found \equiv (\exists\, i, j :: A.i.j = K))$
```

Unfortunately, this program raises an uncaught exception in that the variable *found* is set to true inside, as it violates the invariants. There could be several ways of fixing this without changing the invariants. The exception is raised only when K is found in A. Let us code this into the program using the exception rule so we have

```
{ $x = A$ } $m := 0$; $found := false$;
try
    { inv: $x = A \wedge (\forall\, i, j : 0 \le i < m \wedge 0 \le j < N : A.i.j \ne K) \wedge 0 \le m \le M \wedge \neg found$
      vf: $M-m$ }
    do $m \ne M \rightarrow$ $n := 0$;
        { inv: $x = A \wedge (\forall\, i, j : 0 \le i < m \wedge 0 \le j < n : A.i.j \ne K) \wedge 0 \le m \le M$
                                                    $\wedge 0 \le n \le N \wedge \neg found$
          vf: $N-n$ }
        do $n \ne N \rightarrow$
            " compare A.m.n with K "
            if $x.m.n = K \rightarrow$ raise FoundException()
```

```
                ▯ $x.m.n \neq K \rightarrow$ skip
                fi;  $n \coloneqq n + 1$
            od;  $m \coloneqq m + 1$
        od
  { $x = A \land m = M \land (\forall i, j : 0 \le i < m \land 0 \le j < N : A.i.j \neq K) \land \neg$ found }
  catch FoundException ()
                “ do found ”
                found $\coloneqq$ true
  yrt;
  { $x = A \land$ (found $\equiv (\exists i, j :: A.i.j = K)$)
```

Since we decide not to pass anything when FoundException is raised, our proof obligations reduce to the following two to verify.

- { P } TryBody(raise FoundException() $\leftarrow$ miracle Q) { R }
- { Q } found $\coloneqq$ true { R }

We can then take the following

$P \triangleq x = A \land (\forall i, j : 0 \le i < m \land 0 \le j < N : A.i.j \neq K) \land 0 \le m \le M \land \neg \text{found}$

$Q \triangleq x = A \land (\exists i, j :: A.i.j = K)$

$R \triangleq x = A \land (\text{found} \equiv (\exists i, j :: A.i.j = K))$

Note that TryBody is a repetition so we can use its invariant to be the P in our proof rule. We leave the formal check to the interested reader but make a point that everything is developed in the currently practiced programming methodology – a huge inheritance we can now take for granted as no underlying foundation is changed.

6 Concluding Remarks

We have provided proof rules of exceptions based on Clint and Hoare’s goto rules published decades ago by employing a miraculous statement in a novel manner. It turns out with the discovery of goto’s wp that those rules are extremely simple and easy to understand without any need to amend the underlying proof system. It can readily be extended to allowing multiple catches provided that there are no crossing raise statements, which would otherwise need backward jumps – much more complex constructs. Similarly, we can add a finally clause.

Some work uses traces [9] to describe an exception behavior. We think that their approach could be very operational and is fundamentally different from ours. The work done by King and Morgan [8] falls into a trap to add an extra postcondition for the *exceptional* exit which might not exist in a plausible computation. They intended but failed to make any enhancements over [5], which falls into the same trap of adding extra postconditions. They had wrongly claimed that wp(skip ▯ exit, T) = F, i.e. there would be no guarantee that skip [] exit would terminate. Actually, skip [] exit will terminate, and the termination weakest precondition should be disjunction of the weakest precondition of the following statement and at the end of the corresponding exception block, which most likely might not be F. More importantly, by changing the underlying model, their approach has to amend the entire currently being practiced methodology. Besides, [8] is also limited in scoping. It is unclear why one would still take its complex and costly but incomplete logic now that we have viable alternatives.

7 Acknowledgements

Thanks go to anonymous reviewers for their valuable comments and endorsements.

References

1. Chen, W.: “Loop invariance with break and continue”, Science of Computer Programming, Vol 209, 2021, https://doi.org/10.1016/j.scico.2021.102679.
2. Chen, W.: “Behind Clint and Hoare’s goto Proof Rule”, Proceedings of the 15th International Symposium on Theoretical Aspects of Software Engineering (TASE) 2021, Shanghai, China
3. Chen, W.: “A wp Characterization of Jump Statements”, Proceedings of the 15th International Symposium on Theoretical Aspects of Software Engineering (TASE) 2021, Shanghai, China
4. Clint, M. and Hoare, C. A. R.: “Program proving: Jumps and functions”, Acta Informatica, Vol 1, No 3, September 1972, pp 214-224. https://doi.org/10.1007/BF00288686
5. Cristian, F.: “Correct and Robust Programs”, IEEE Transaction on Software Engineering, Vol 10, No 2, March 1984, pp 163-174.
6. Dijkstra, E.W.: “A Discipline of Programming”, Prentice-Hall, Englewood Cliffs, 1976
7. Dijkstra, E.W. and van Gasteren, A.J.M.: “A simple fixed-point argument without the restriction to continuity”, Avg 45a/EWD 901a, October 1985
8. King, S. and Morgan, C.: “Exits in the refinement calculus”, Formal Aspects of Computing Vol 7, 1995, pp 54–76. https://doi.org/10.1007/BF01214623
9. Leino, K.R.M. and van de Snepscheut, J.L.A.: “Semantics of Exceptions”, JAN-185, October 1993.
10. O'Donnell, M. J.: “A critique of the foundations of Hoare style programming logics”, Communnications of ACM, Vol 25, No 12, December 1982, pp 927-935. https://doi.org/10.1145/358728.358748