Automatic Program Generation Using Sequent Calculus

Talal Maghrabi
Department of Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287-5406

Forouzan Golshani
Bull Worldwide Information Systems
Phoenix, Arizona 85060
(on sabbatical leave from ASU)

Abstract

Program development can be made amenable to formal methods by using a logical framework. A logic specification, whose operational semantics is based on proof theory, provides an abstract and "implementation independent" definition of the problem, the data domains and the associated operators.

Unlike many of the current efforts in this area that use resolution, our approach is based on natural deduction, more specifically, sequent calculus. Following the methodology proposed by Manna and Waldinger, we propose the synthesis tableau technique by which we construct a proof for the well formed formula representing the specification. The desired program is obtained as a side effect of the proof process.

1. Introduction

Automatic Program Synthesis (or APS), also known as automatic programming, is the process of using computers to systematically generate programs, or parts of programs, from given specifications. Research in program synthesis began around the period of developing the early compilers. It was motivated by the need to simplify programming. As it evolved, the motivation was shifted to the generation of computer programs in more reliable ways.

One of the interesting approaches to program synthesis is based on theorem proving. Informally speaking, automated theorem proving is the process of using a computer to prove, or help in proving, theorems. In this approach the specifications of the desired program is given as a well-formed formula (wff) in First Order Logic (FOL). A theorem prover will prove the existence of such a program by showing that this wff is a theorem. The program may be generated as a side-effect of this process.

There are now a number of prototype systems that implement this approach. Typically, they use various versions of resolution [Wos84] as their deduction mechanism. The main drawback of resolution theorem proving is that the proofs generated are not constructive, i.e. a wff can be shown to be a theorem but no constructive proof is provided.

On the other hand, several interactive theorem provers based on natural deduction have been developed. *Natural deduction*, and hence *sequent calculus*, produces proofs in a constructive manner. The proofs are constructed in a way that is close to human "natural" reasoning. These proofs can be easily followed, and modified if necessary. When a natural deduction based interactive theorem prover fails to prove a given wff, the user can check the proof and modify it or redirect the prover when needed. This interaction between the user and the prover is very essential when dealing with missing information, which is commonly found with programs specifications.

Research in this field is important since using machines to generate programs from specifications may have the impact, at least in the long run, of reducing the cost and time of the software life cycle. Moreover, using automatic theorem provers to generate programs will guarantee that the generated software will fulfill the original specification since it was generated based on a sound mathematical proof [Gren69, Mann80,Wald69].

This paper describes an attempt toward developing a methodology, called the *synthesis tableau*, for generating programs from natural deduction proofs. The motivation behind using natural, rather than classical, deduction is that natural deduction proofs are conducted in a more intuitive way. In the cases of failure, the user can check the process and modify or redirect the prover as needed.

The paper concentrates on two aspects of our research on automated synthesis of programs: (1) defining the target language in which the desired programs will be constructed; and (2) developing a method that maps the steps of a constructed proof to the corresponding statements in the target language. The target language is a primitive Prolog-like language in which programs are represented by a set of statements. Each statement is defined on a set of variables which are either input or output. The synthesis method utilizes a tableau, called the synthesis tableau, similar to the tableau of Manna and Waldinger [Mann80, Mann85].

In section 2, we describe automatic program synthesis. Section 3 outlines the proposed method. Section 4 contains an example that demonstrates the method.

2. Automatic Program Synthesis - An Overview

Rapid developments in the field of computer programming resulted in much interest in *Automatic Programming* (AP) or automatic program synthesis. AP simply states that computers can be instructed to generate programs for performing certain tasks in an executable language from some given specification [Barr82, Bier76].

In the early investigations, the major motivation for AP was to simplify programming and relieve programmers from handling so many details. In fact, compilers were originally developed as automatic programming systems. Nowadays the major goal of AP is a different one. It may be a long time, if ever, before large programs can be generated automatically, but the current goal of AP is to develop programs using more reliable methodologies. Current software development methods are costly and generally unreliable. Much of the delivered software fails to satisfy its specification [Schu90].

Several frameworks for automatic program synthesis exist, including theorem proving and program-transformation [Barr82]. The theorem proving approach is described in the next section. In the program-transformation approach, the original specification S is transformed through several steps until it evolves into a set of statements in some executable language. Typically, specifications are written in a more powerful logic (possibly, higher order logics), and the steps of transformation reduce the formulae into statements in a less powerful logic that is more suitable for computation. This is motivated by the fact that higher order logics have more expressive power but are less suited for computational purposes. On the other hand, lower level logics, such as Horn Clause logic, are computationally efficient but are less expressive [Gols88]. The following diagram illustrates this spectrum:

Horn Clause Logic FOL Higher order logic
<---------->
More efficiently executable More expressive

Dijkstra developed an elegant and interesting methodology for generation of programs and their proof of correctness [Cohn90, Dijk76]. In his approach, programs are constructed using a set of preconditions and post-conditions that must be satisfied by the program. Suppose a program S must start at an initial state satisfying the input condition Q and must terminate in a final state satisfying the output condition R. Using Hoare's notation [Gold82], this program is represented as {Q}S{R}. The target language of Dijkstra is a simple language with several constructs such as selection, repetition, and assignment. Preconditions and post-conditions are defined precisely for each construct. Both Q and R are used to generate program S as a set of constructs in the target language.

2.1 Program Synthesis Using Theorem Proving

The idea of using computers to generate programs directly from the specification using mathematical methods is one of the most interesting and promising ideas in the field of computer science. Programming, as viewed by some mathematicians, can be considered as a mathematical activity in which programs are considered as complex mathematical expressions. This means that a program can be derived directly from its specification [Drom89].

The synthesis of programs using theorem proving is a process that consists of two major parts: a *proof* and a *program*. The proof is the sequence of steps followed to show that, based on the available axioms and facts of some theory, the given wff (specification) is a theorem in that theory. The program is a sequence of statements, in some target language, that when executed with the given input produces the correct output that meets the given specification.

The specification of a program allows us to express the purpose of the program without showing how this purpose can be achieved. To use theorem proving in synthesizing programs, specifications of programs need to be stated in a way suitable for theorem provers. Generally, a specification of a program can be described as follows [Mann80]:

$$Prog(x) \Leftarrow \text{find y such that } O(x,y) \text{ where } I(x)$$

Here, x denotes the input to the desired program; y denotes its output; I(x) represents the input condition; and O(x,y) represents the output conditions to be satisfied by Prog(x). In other words, we are attempting to construct a program, Prog, such that for an arbitrary input x satisfying the input condition I(x) the output Prog(x) satisfies the output condition O(x,Prog(x)).

For example, to specify a program that computes the integer square root of a non-negative integer n, we would write [Mann80]:

$$sqrt(n) \Leftarrow \text{find y such that } \text{integer}(y) \text{ and } y^2 \le n < (y+1)^2 \text{ where } \text{integer}(n) \text{ and } 0 \le n.$$

In using theorem proving to derive a program from such a specification, we are actually attempting to prove a theorem of the form:

$$\forall x\, (I(x) \rightarrow \exists y\, O(x,y))$$

This well-formed formula (wff) is read as follows: for all input x, if x satisfies the condition I(x), then there exists an output y such that the condition O(x,y) is satisfied.

The specification of the integer square root program using FOL would be:

$$\forall n\, (\text{integer}(n) \wedge 0 \le n \rightarrow \exists y\, \text{integer}(y) \wedge y^2 \le n < (y+1)^2)$$

The existence of such a program can be shown by proving that the above wff is a theorem of a formal system, in this case FOL. The desired program will be generated as a side effect of this process. PROW [Wald69] and QA3 [Gren69] are examples of this approach. These systems were based on resolution theorem proving. The major disadvantage of resolution-based theorem provers is that

when they fail to prove a given wff, generally, the user cannot interfere in the proving process. Even when the prover is interactive, the proof generated is not constructive, and thus is difficult to follow and/or modify.

As mentioned before, several other theorem provers are based on natural deduction [Bled83, Paul87] whose proofs can be followed by the user, and hence can be corrected or modified when needed. Although some researchers have attempted to generate programs by using a theorem prover based on natural deduction [Goto78, Goto79, Sato79], these attempts have been limited to the domain of natural numbers only, and have not been shown to be effective in other domains.

2.2 Natural Deduction Theorem Proving

Natural deduction is a deduction mechanism that can be used with the languages of propositional or first-order predicate calculi to derive wffs in a *seemingly* natural way. This mechanism was originally formalized by Gentzen [Szab69]. The original motivation was to set up an intuitive formal system. Later, Gentzen formulated *sequent calculus* in which theorems are proved using natural deduction, and in which proofs are independent of assumptions.

In natural deduction, each logical connective or quantifier is defined by two rules: introduction and elimination. The introduction rule of a connective, say $\vee$, determines when $\mathcal{A} \vee \mathcal{B}$ can be concluded. On the other hand, the elimination rule determines what can be concluded from $\mathcal{A} \vee \mathcal{B}$ [Paul87]. Note that $\mathcal{A}$ and $\mathcal{B}$ are syntactic meta symbols that can be used to represent any wff. A proof in natural deduction may be seen as a collection of formulas arranged in a tree form in which the theorem is at the root, arcs are labeled with inference rules, and leaves are axioms and/or assumptions.

In sequent calculus each wff has its own truth without being based on any assumptions. The building block of sequent calculus is *sequent* . A sequent has the form:

$$\mathcal{A}_1, \ldots, \mathcal{A}_n \vdash \mathcal{B}$$

where $\{\mathcal{A}_1,\ldots,\mathcal{A}_n\}$ is called the *antecedent* and $\mathcal{B}$ is called the *consequence*. This sequent states that the wff $\mathcal{B}$ is provable from the set of assumptions $\mathcal{A}_1,\ldots,\mathcal{A}_n$. Informally, the sequent $\mathcal{A}_1,\ldots, \mathcal{A}_n \vdash \mathcal{B}$ means that $\mathcal{B}$ is true whenever each and every $\mathcal{A}_i$ is true.

In sequent calculus, inference rules are defined differently than in natural deduction. Each connective or quantifier is defined by two rules depending on its position with respect to $\vdash$. A *left rule* of a connective (or a quantifier) determines what should be concluded when that connective is on the left of $\vdash$, while the *right rule* determines what should be concluded when the connective is on the right of $\vdash$. The left rules are similar to the elimination rules, and the right rules are similar to the introduction rules of natural deduction. For a complete and comprehensive discussion of sequent calculus the reader may refer to [Paul87, Szab69]. A few of the inference rules of sequent calculus are shown in Figure 1. In these rules, Γ and Δ represent sets of wffs; the $\Gamma, \mathcal{A}$ represents the union of Γ and $\{\mathcal{A}\}$; and $\mathcal{A}[t/x]$ means that each occurrence of the variable x in $\mathcal{A}$ is replaced by the term t. The rule $\dfrac{\Gamma \vdash \mathcal{A}}{\Delta \vdash \mathcal{B}}$ is read bottom-up as follows: given the sequent $\Delta \vdash \mathcal{B}$, conclude the sequent $\Gamma \vdash \mathcal{A}$.

3. The Synthesis Methodology

In this research two major components are needed: a *target language* in which programs will be constructed, and a *construction method* in which both proofs and programs are generated. Our target language, called L_s, is a simple Prolog-like language in which programs are constructed using predicates and functions. The construction method we have developed here, called the *synthesis tableau*, is similar to the method of Manna and Waldinger since both of them uses a tableau to represent the proof and the program construction. This method is, however, different from theirs in the proving methodology. While Manna and Waldinger used some versions of resolution to conduct the proof, we rely completely on sequent calculus.

This section first presents an informal description of our method. Essentially, a program is synthesized by constructing a proof in sequent calculus for the wff representing the given specification along with developing statements in the target language that represent the desired program. The completion of the proof signifies the end of program construction.

3.1 The Target Language

The target language L_s is a small subset of the language of Horn clauses. The motivation behind choosing such a simple language is that we needed a language that demonstrates our construction method without the burden of using the various constructs of a complex language.

A program consists of one or more statements (Horn clauses). Each statement, in the program, consists of one or more predicates. The ":-" is used to represent conditional statements, i.e. $p_1(t_1)$:- $p_2(t_1,t_2)$ means that the 1-place predicate $p_1(t_1)$ is true if the 2-place predicate $p_2(t_1,t_2)$ is true. The "," is used to represent conjunction of predicates in *conditional* statements, i.e. $p_1(t_1)$:- $p_2(t_1,t_2)$, $p_3(t_3)$ means that $p_1(t_1)$ is true if *both* $p_2(t_1,t_2)$ and $p_3(t_3)$ are true.

Suppose x_1,x_2,x_3 are non-negative integers, and suppose $p_1(x_1,x_2,x_3)$ stands for "x_3 is the quotient of dividing x_1 by x_2", and $p_2(x_1,x_2)$ stands for "x_1 is less than x_2". Then the expression "if $x_1 < x_2$ then $x_1/x_2 = 0$" is represented by the following statement in L_s:

$$p_1(x_1,x_2,0) \text{ :- } p_2(x_1,x_2)$$

	Right	Left
$\vee$:	$\frac{\Gamma \vdash \mathcal{A}}{\Gamma \vdash \mathcal{A} \vee \mathcal{B}}$ $\frac{\Gamma \vdash \mathcal{B}}{\Gamma \vdash \mathcal{A} \vee \mathcal{B}}$	$\frac{\Gamma, \mathcal{A} \vdash C \quad \Gamma, \mathcal{B} \vdash C}{\Gamma, \mathcal{A} \vee \mathcal{B} \vdash C}$
$\rightarrow$:	$\frac{\Gamma, \mathcal{A} \vdash \mathcal{B}}{\Gamma \vdash \mathcal{A} \rightarrow \mathcal{B}}$	$\frac{\Gamma \vdash \mathcal{A} \quad \Gamma, \mathcal{B} \vdash C}{\Gamma, \mathcal{A} \rightarrow \mathcal{B} \vdash C}$
$\forall$:	$\frac{\Gamma \vdash \mathcal{A}}{\Gamma \vdash \forall x.\mathcal{A}}$ (*x is not free in* Γ)	$\frac{\Gamma, \mathcal{A}[t/x] \vdash \mathcal{B}}{\Gamma, \forall x.\mathcal{A} \vdash \mathcal{B}}$
$\exists$:	$\frac{\Gamma \vdash \mathcal{A}[t/x]}{\Gamma \vdash \exists x.\mathcal{A}}$	$\frac{\Gamma, \mathcal{A} \vdash \mathcal{B}}{\Gamma, \exists x.\mathcal{A} \vdash \mathcal{B}}$ (*x is not free in* Γ *or* $\mathcal{B}$)

Figure 1 : Sequent Calculus Inference Rules

Proof	Program
(i) $\mathcal{A} \vdash \mathcal{B}$	$\mathcal{P}_j$

(i) is used to number the proof step, $\mathcal{A}$ represents the assumptions of the proof step, $\mathcal{B}$ represents the conclusion of the proof step, and $\mathcal{P}_j$ represents some statement(s) in the target language.

Figure 2 : A Typical Row in The Synthesis Tableau

Proof	Program
(1) $\vdash \forall x\ I(x) \rightarrow \exists y\ O(x, y)$	$P_1(x,y)$

Figure 3 : The Initial Tableau

3.2 The Basic Structure

The basic structure of our method is a tableau called the *synthesis tableau*. This tableau, which is similar to (but different from) the tableau of Manna and Waldinger [Mann80], consists of two columns called *proof* and *program*. Each row contains the proof step in the proof column and the segment of the program generated by that step of the proof in the program column. Obviously, the entries to the program column have no effect on the proof process. A typical row in the tableau is shown in Figure 2. Note that the program entry may be empty.

Proofs are conducted using sequent calculus with backward chaining. We start with the given wff, and use the left or right inference rules of connectives (or quanitifiers) to reduce the given sequent into one or more new sequents. These new sequents may be reduced further. This process is repeated until we end up only with axioms. For example, in order to prove the sequent $\Gamma \vdash \forall x\ \mathcal{A}$ it is sufficient to reduce it into the sequent $\Gamma \vdash \mathcal{A}$ and then prove this new sequent.

In our early attempts, as described in [Magh91], the proofs were conducted using sequent calculus with forward chaining, i.e. we started with the axioms and used the inference rules to derive the given wff. We, also, used separate columns to represent the assumptions (antecedents) and the conclusion (consequence) of the sequent. However, it became clear that the separation of these columns was not necessary, and the proof step could be represented as a single column. It also became clear that forward proofs were not appropriate. Assigning the initial predicate to an axiom was incorrect since it violated our basic rule which states that each program entry must satisfy the corresponding proof step. Therefore, the initial predicate should only be assigned to the original wff (i.e. the specification).

Suppose we are given the original wff (specification) as $\forall x\ I(x) \rightarrow \exists y\ O(x, y)$, and suppose we choose $P_1(x,y)$ to be the initial (or main) predicate of the target program, then the initial tableau will be as shown in Figure 3.

3.3 Program Construction

Given a wff representing the specification, the desired program may be generated as follows:

1) A suitable initial predicate is assigned to the program entry of the first row in the tableau. By suitable we mean that the predicate has the appropriate input and output variables. The initial tableau will be similar to the one shown above.
2) The proof is done by backward chaining as stated earlier.
3) During the derivation of the proof, the program entries in the tableau are refined in the following manner:
   * The creation of a new proof step (sub-goal) may result in a creation of a new suitable predicate.
   * All substitutions of variables made in the proof process are also applied to the program entry.
   * The program entry of a new sub-goal is derived from the program entry of its original goal using the rules which will be described in section 3.4.
4) The process terminates upon completion of proofs of all sub-wffs. The desired program will then be the collection of the program entries of each sub-goal.

The above procedure is entirely different from that of Manna and Waldinger. First, in their method they separated the assumptions and conclusion of the sequent into two columns: goals and assertions, and no row in the tableau can have both a goal and an assertion. Second, they applied some versions of resolution between goals and/or assertions, thus they do not use natural deduction. Finally, their process terminates when a false assertion (or a true goal) is reached.

3.4 Inference Rules and Program Entries

Since each new step in a proof is generated by applying an inference rule, we need to define how the application of such inference rules affects the associated program entries. We should note, however, that while the proof of the sub-goal(s) suffices as a proof of the original goal, the program associated with the original goal depends on the program(s) associated with the sub-goal(s). Thus all program entries should be kept during the construction process. Below we will present some of the inference rules. A complete list can be found in [Magh92].

- $\forall$-right Inference rule

The $\forall$-right inference rule is $\dfrac{\Gamma \vdash \mathcal{A}}{\Gamma \vdash \forall x\ \mathcal{A}}$. A backward reading of this rule states that in order to prove the sequent $\Gamma \vdash \forall x\ \mathcal{A}$, it is sufficient to prove the sequent $\Gamma \vdash \mathcal{A}$ provided that x is not free in Γ.

Assume that during our construction process we obtained the row shown in Figure 4.a, where $\mathcal{P}_j$ is a statement in L_s. By the above inference rule we can conclude that since the proof of the sequent $\Gamma \vdash \mathcal{A}$ leads to a proof of the sequent $\Gamma \vdash \forall x\ \mathcal{A}$, then the sequent $\Gamma \vdash \forall x\ \mathcal{A}$ can be replaced by the sequent $\Gamma \vdash \mathcal{A}$ to generate the a new entry in the proof column.

In the program column, since the only change made to the proof entry is removing the quantifier, which does not change the original specification, the new row in the tableau should have the same program entry, i.e. the new row in the tableau should be as shown in Figure 4.b.

- $\exists$-right Inference rule

The $\exists$-right inference rule is $\dfrac{\Gamma \vdash \mathcal{A}[t/x]}{\Gamma \vdash \exists x\ \mathcal{A}}$. This rule states that proving the sequent $\Gamma \vdash \mathcal{A}[t/x]$ is sufficient for the proof of the sequent $\Gamma \vdash \exists x\ \mathcal{A}$. Using the same reasoning as in the first rule, we find out that the row shown in Figure 5.a can be replaced by the row shown in Figure 5.b. Note that $\mathcal{P}_j[t/x]$ is obtained from $\mathcal{P}_j$ by replacing each occurrence of the variable x by the term t.

- $\rightarrow$-right Inference rule

The $\rightarrow$-right inference rule is $\dfrac{\Gamma, \mathcal{A} \vdash \mathcal{B}}{\Gamma \vdash \mathcal{A} \rightarrow \mathcal{B}}$. This rule states that in order to prove the sequent $\Gamma \vdash \mathcal{A} \rightarrow \mathcal{B}$ we need a proof of the sequent $\Gamma, \mathcal{A} \vdash \mathcal{B}$. If we obtain the row shown in Figure 6.a. then we can replace it with the row shown in Figure 6.b.

- $\vee$-left Inference rule

The $\vee$-left inference rule is $\dfrac{\Gamma, \mathcal{A} \vdash C \quad \Gamma, \mathcal{B} \vdash C}{\Gamma, \mathcal{A} \vee \mathcal{B} \vdash C}$. The rule states that the proof of the sequent $\Gamma, \mathcal{A} \vee \mathcal{B} \vdash C$ can be achieved by proving both of the sequents $\Gamma, \mathcal{A} \vdash C$ and $\Gamma, \mathcal{B} \vdash C$. This rule is different from the previous rules since it reduces the original goal to two subgoals. Suppose, during the synthesis of a program, we generate the row shown in Figure 7.a, then, we need to indicate that $\mathcal{P}_j$ depends on both the programs generated by the new rows, say $\mathcal{P}_{j1}$ and $\mathcal{P}_{j2}$, based on $\mathcal{A}$ or $\mathcal{B}$ respectively.
So if the proof of the new subgoals generate the rows shown in Figure 7.b, we will have to rewrite the program entry in step (i) as :

$$\mathcal{P}_j :- \mathcal{P}_{i1}, \mathcal{P}_{j1}$$
$$\mathcal{P}_j :- \mathcal{P}_{i2}, \mathcal{P}_{j2}$$

where $\mathcal{P}_{i1}$ and $\mathcal{P}_{i2}$ are the representation of $\mathcal{A}$ and $\mathcal{B}$ respectively, in the target language.

4. An Example

Assume that we want to generate a program that computes the integer quotient and remainder of dividing two integers. The problem can be specified as follows:

(i) $\Gamma \vdash \forall x\, \mathcal{A}$	$\mathcal{P}_i$

Figure 4.a : A row with $\Gamma \vdash \forall x\, \mathcal{A}$ in the proof column.

(i+1) $\Gamma \vdash \mathcal{A}$	$\mathcal{P}_i$

Figure 4.b : The above row after the application of $\forall$-right rule.

(i) $\Gamma \vdash \exists x\, \mathcal{A}$	$\mathcal{P}_i$

Figure 5.a : A row with $\Gamma \vdash \exists x\, \mathcal{A}$ in the proof column.

(i+1) $\Gamma \vdash \mathcal{A}[t/x]$	$\mathcal{P}_i[t/x]$

Figure 5.b : The above row after the application of $\exists$-right rule.

(i) $\Gamma \vdash \mathcal{A} \rightarrow \mathcal{B}$	$\mathcal{P}_i$

Figure 6.a : A row with $\Gamma \vdash \mathcal{A} \rightarrow \mathcal{B}$ in the proof column.

(i+1) $\Gamma, \mathcal{A} \vdash \mathcal{B}$	$\mathcal{P}_i$

Figure 6.b : The above row after the application of $\rightarrow$-right rule.

(i) $\Gamma, \mathcal{A} \vee \mathcal{B} \vdash C$	$\mathcal{P}_i$

Figure 7.a : A row with $\Gamma, \mathcal{A} \vee \mathcal{B} \vdash C$ in the proof column.

(i+1) $\Gamma, \mathcal{A} \vdash C$	$\mathcal{P}_{i1}$
(i+2) $\Gamma, \mathcal{B} \vdash C$	$\mathcal{P}_{i2}$

Figure 7.b : The new rows associated with the new proof steps.

$$\forall x_1 x_2\ (x_1 \geq 0 \wedge x_2 > 0 \rightarrow \exists y_1 y_2\ x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * y_1 + y_2) \qquad (S)$$

Here, we use the proof strategy in [Cons78]. We start by choosing the predicate $p_1(x_1,x_2,y_1,y_2)$ as the initial (or main) predicate, and stores it in the program column of the first row in the tableau. The proof column has the initial specification (as a sequent). Thus, the initial tableau is shown in Figure 8.a.

By applying the rules presented in the previous section, we reduce the original sequent to a new one obtaining the the row shown in Figure 8.b. From number theory, we know that the expression $x_1 \geq 0 \wedge x_2 > 0$ can be replaced by the expression $x_1 < x_2 \vee x_1 \geq x_2$. This transformation will not change the program entry. Thus we obtain the row shown in Figure 8.c.

Using the $\vee$-left inference rule of the previous section, and if $p_2(x_1,x_2)$ denotes the relation $x_1 < x_2$, and $p_3(x_1,x_2)$ denotes the relation $x_1 \geq x_2$ (in the target language), we generate the row shown in Figure 8.d.

Now we have two subgoals whose antecedents are $x_1 < x_2$ and $x_1 \geq x_2$, respectively. The first subgoal $x_1 < x_2 \vdash x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * y_1 + x_2$ (whose program entry has the predicate $p_4(x_1,x_2,y_1,y_2)$) is proved as shown in Figure 8.e.

Since $x_1 < x_2$, we conclude that for $x_1 = x_2 * y_1 + y_2$ to be true we must assign the value 0 for y_1 and the value x_1 for y_2. Let the constant symbol a_1 of the target language denotes 0, we obtain the row shown in Figure 8.f.

The second subgoal $x_1 \geq x_2 \vdash x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * y_1 + x_2$ (whose program entry has the predicate $p_5(x_1,x_2,y_1,y_2)$) is proved as shown in Figure 8.g.

The proof here is done by complete induction on x_1. Let $q(x_1,x_2)$ represents the quotient of x_1/x_2 and $r(x_1,x_2)$ represents the remainder of x_1/x_2. The concept of complete induction states that in order to prove

$$x_1 \geq x_2 \wedge x_2 > 0 \rightarrow x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * q(x_1,x_2) + r(x_1,x_2)$$

we assume the induction hypothesis, which states the above wff holds for all x' that are less than x_1. i.e.

$$x' < x_1 \rightarrow (x' \geq x_2 \wedge x_2 > 0 \rightarrow x_2 > y_2 \wedge y_2 \geq 0 \wedge x' = x_2 * q(x',x_2) + r(x',x_2))$$

Now, by setting x' to $x_1 - x_2$ in the induction hypothesis, the induction proof introduces a recursive call to the remainder-quotient program with x_1 replaced by $x_1 - x_2$. This is shown in Figure 8.h.

Using arithmetic manipulation and renaming the variables $q(x_1 - x_2, x_2)$ and $r(x_1 - x_2, x_2)$ to y_1 and y_2, respectively, and if $f_1(y_1,1)$ denotes the expression $y_1 + 1$ then we obtain the final row as shown in Figure 8.i.

By proving the subgoals in (5) and (7), we have proven the original goal in (4), and hence we have proven the original wff in (S). Therefore, the desired program that satisfies the given specification for the quotient and remainder of non-negative integers is:

$p_1(x_1,x_2,y_1,y_2)$:- $p_2(x_1,x_2)$, $p_4(x_1,x_2,y_1,y_2)$
$p_1(x_1,x_2,y_1,y_2)$:- $p_3(x_1,x_2)$, $p_5(x_1,x_2,y_1,y_2)$
$p_4(x_1,x_2,a_1,x_1)$
$p_5(x_1,x_2,f_1(y_1,1),y_2)$:- $p_1(x_1 - x_2,x_2,y_1,y_2)$

Proof	Program
(1) $\vdash \forall x_1 x_2\ (x_1 \geq 0 \wedge x_2 > 0 \rightarrow \exists y_1 y_2\ x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * y_1 + y_2)$	$p_1(x_1,x_2,y_1,y_2)$.

Figure 8.a : The Initial tableau for the example

(2) $x_1 \geq 0 \wedge x_2 > 0 \vdash x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * y_1 + y_2$	$p_1(x_1,x_2,y_1,y_2)$.

Figure 8.b : The tableau after applying some inference rules

(3) $x_1 < x_2 \vee x_1 \geq x_2 \vdash x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * y_1 + y_2$	$p_1(x_1,x_2,y_1,y_2)$.

Figure 8.c : The tableau after applying some transformation rules

(4) $x_1 < x_2 \vee x_1 \geq x_2 \vdash x_2 > y_2 \wedge y_2 \geq 0 \wedge x_1 = x_2 * y_1 + y_2$	$p_1(x_1,x_2,y_1,y_2)$:- $p_2(x_1,x_2)$, $p_4(x_1,x_2,y_1,y_2)$. $p_1(x_1,x_2,y_1,y_2)$:- $p_3(x_1,x_2)$ $p_5(x_1,x_2,y_1,y_2)$.

Figure 8.d : The tableau after applying the $\vee$-left inference rule

(5) $x_1<x_2 \vdash x_2>y_2 \wedge y_2 \geq 0 \wedge x_1=x_2*y_1+y_2$	$p_4(x_1,x_2,y_1,y_2)$.

Figure 8.e The row generated for proving the first sub-goal.

(6) $x_1<x_2 \vdash x_2>y_2 \wedge y_2 \geq 0 \wedge x_1=x_2*y_1+y_2$	$p_4(x_1,x_2,a_1,x_1)$

Figure 8.f The row that shows the proof of the first sub-goal.

(7) $x_1 \geq x_2 \vdash x_2>y_2 \wedge y_2 \geq 0 \wedge x_1=x_2*y_1+x_2$	$p_5(x_1,x_2,y_1,y_2)$

Figure 8.g The row generated for proving the second sub-goal.

(8) $x_1-x_2 \geq x_2 \wedge x_2>0 \rightarrow x_2>y_2 \wedge y_2 \geq 0 \wedge$ $x_1-x_2 = x_2 * q(x_1-x_2,x_2) + r(x_1-x_2,x_2)$	$p_5(x_1,x_2,y_1,y_2)$:- $p_1(x_1-x_2,x_2,y_1,y_2)$

Figure 8.h The row generated by the induction hypothesis

(9) $x_1 \geq x_2 \vdash x_2>y_2 \wedge y_2 \geq 0 \wedge x_1=x_2*y_1+x_2$	$p_5(x_1,x_2,f_1(y_1,1),y_2)$:- $p_1(x_1-x_2,x_2,y_1,y_2)$

Figure 8.i The row that shows the proof of the second sub-goal.

This example demonstrates how a program can be constructed as a side-effect of a natural deduction proof.More examples, particularly those involving other rules of inference, are presented in [Magh92].

5. Conclusion

Modern software engineering techniques aim at developing programs that are shorter, clearer and, above all, correct. The well-recognized "software crisis" is a symptom of the limitations inherent in the traditional approach to the specification, design and programming of complex systems.

Although programming languages have not changed a great deal in the past decade, programming methodologies have seen considerable changes. Investigations into program correctness have led to the discovery of several novel and interesting methods for program development. In this research, we have pursued the idea that programs and their proof of correctness may be generated hand-in-hand within a rigorous formal framework. Several techniques, such as Dijkstra's program proving framework, program transformation techniques, and the tableau methods, are developed based on this ideology.

We have chosen the tableau method, and have developed a framework for program derivation called the synthesis tableau. Sequent calculus is used for construction of proofs and derivation of programs. It is important to note that, in this methodology, the proof ideas lead the way for the generation of program. If applied carefully, the derived program will (provably) adhere to its specification, and will be free of error. By using this rigorous framework, we will have "correctness by construction" instead of costly "postmortem" verifications which, in most cases, are not adequate.

References

[Barr82] A. Barr and E. Feigenbaun (Eds), *The Handbook of Artificial Intelligence, Vol.* 2, Chapter X, William Kaufmann Inc., pp. 297-325, 1982.

[Bier76] A. Biermann, "Approaches to Automatic Programming," in *Advances in Computers*, Vol. 15, pp. 1-63, Academic Press, 1976.

[Bled83] W. Bledsoe, "The UT Prover," *Math Department Memo ATP-17B*, University of Texas At Austin, April 1983.

[Cohn90] E. Cohen, *Programming in the 1990s: An Introduction to The Calculation of Programs*, Springer-Verlag, 1990.

[Cons78] R. Constable and M. O'Donnell, *A Programming Logic with An Introduction to The PL/CV Verifier*, Winthrop Publisher, Inc., 1978.

[Dijk76] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Drom89] G. Dromey, *Program Derivation: The Development of Programs from Specifications*, Addison-Wesley, 1989.

[Gold82] R. Goldblatt, "Axiomatising The Logic Of Computer Programming," *Lecture Notes in Computer Science*, Vol. 130, Springer-Verlag, 1982.

[Gols88] F. Golshani, W. Scott and P. White, "Languages for Intelligent Specification Systems," *Proc. of IEEE International Conference on Computer Languages*, Miami Beach, FL, pp 304-311, 1988.

[Goto78] S. Goto, "Program Synthesis through Godel's Interpretation," *Proc. of the* Int. *Conference on Mathematical Studies of Information Processing*, pp. 302-325, 1978.

[Goto79] S. Goto, "Program Synthesis from Natural Deduction Proofs," *proc. of Int.Joint Conference on Artificial Intelligence*, pp. 339-341, 1979.

[Gren69] C. Green, "Application of Theorem Proving to Problem Solving," *proc. of IJCAI*, pp. 219-239, 1969.

[Magh91] T. Maghrabi, "Generating Programs from Natural Deduction Proofs," *Proc. of The First Golden West Conference on Intelligence Systems*, Reno, Nevada, pp. 150-156, June 1991.

[Magh92] T. Maghrabi, "The Synthesis Tableau: An Automatic Programming Approach Using Sequent Calculus," *Ph. D dissertation*, Department of Computer Science and Engineering, Arizona State University, 1992.

[Mann80] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis," *ACM Transactions on Programming Languages and Systems*, (2), pp. 90-121, 1980.

[Mann85] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming , Vol. 2*, Addison-Wesley, 1985.

[Paul87] L. Paulson, *Logic and Computation: Interactive Proofs with Cambridge LCF*, Cambridge University Press, 1987.

[Sato79] M. Sato, "Towards A Mathematical Theory of Program Synthesis," *proc. of Int.Joint Conference on Artificial Intelligence*, pp. 757-762, 1979.

[Schu90] G. Schulmeyer, *Zero Defect Software*, McGraw-Hill, 1990.

[Szab69] M. Szabo (Ed.), *The Collected Papers of Gerhard Gentzen*, North-Holland Publishing Company, pp. 68-128, 1969.

[Wald69] R. Waldinger and R. Lee, "PROW: A Step Toward Automatic Program Writing," *proc. of IJCAI*, pp. 241-252, 1969.

[Wos84] L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning: Introduction and Applications*, Prentice-Hall, 1984.