FUNDAMENTALS OF DEDUCTIVE PROGRAM SYNTHESIS

Zohar Manna
Computer Science Department
Stanford University
and
Computer Science Department
Weizmann Institute of Science

Richard Waldinger
Artificial Intelligence Center
SRI International
and
Computer Science Department
Stanford University

Index terms: automated deduction, deductive tableau, formal methods, program synthesis, program transformation, specifications, theorem proving.

Abstract

An informal tutorial is presented for program synthesis, with an emphasis on deductive methods. According to this approach, to construct a program meeting a given specification, we prove the existence of an object meeting the specified conditions. The proof is restricted to be sufficiently constructive, in the sense that, in establishing the existence of the desired output, the proof is forced to indicate a computational method for finding it. That method becomes the basis for a program that can be extracted from the proof.

The exposition is based on the deductive-tableau system, a theorem-proving framework particularly suitable for program synthesis. The system includes a nonclausal resolution rule, facilities for reasoning about equality, and a well-founded induction rule.

INTRODUCTION

This is an introduction to program synthesis, the derivation of a program to meet a given specification. It focuses on the deductive approach, in which the derivation task is regarded as a problem of proving a mathematical theorem.

Let us outline this approach in very general terms. We here construct only applicative (functional) programs. We are given a specification that describes a relation between the input and output of the desired program. The specification does not necessarily suggest any method for computing the output. To construct a program that meets the specification, we prove the existence, for any input object, of an output object that satisfies the specified conditions. The proof is conducted in a background theory that expresses the known properties of the subject domain and describes

This research was supported in part by the National Science Foundation grants CCR–89–04809, CCR–89–11512, and CCR–89–13641, by the Defense Advanced Research Projects Agency under contract NAG2–703, and by the United States Air Force Office of Scientific Research under contract AFOSR–90–0057.

A preliminary version of portions of the manuscript appears in *Logic, Algebra, and Computation* (W. Meixner, ed.), NATO ASI Series, Series F: Computer and Systems Sciences, Springer-Verlag, Berlin, 1991.

the primitives of the programming language. The proof is restricted to be sufficiently constructive that, to establish the existence of a satisfactory output object, it is forced to indicate a computational method for finding one. That method becomes the basis for a program that can be extracted from the proof.

In principle, many theorem-proving methods can be adapted for program synthesis. We have developed a proof system, called the deductive tableau, that is specifically intended for this purpose.

In this paper, we begin by defining program synthesis and relating it to other software development technology. We then introduce the deductive-tableau proof system and show how to extract programs from tableau proofs.

Specifications

Program synthesis begins with a specification; in our case, this is a representation of the relationship between the input and output. A specification should be a description of the purpose or expected behavior of the desired program; ideally, it is close to the intentions of the users of the system. A good specification is clear and readable; we do not care if it describes an efficient computation, or indeed any computation at all. A program, on the other hand, is primarily a description of a computation, preferably an efficient one.

While many languages have been proposed for specification, we have settled on logic in our own work, because it is quite general and appropriate for deductive methods. If other languages are more appropriate for particular subject domains, it is plausible that they be translated into logic.

Let us give logical specifications for some familiar programs.

Example (sorting specification). Suppose we would like our programs to sort a list of numbers. Then we may be given the specification

$$sort(\ell) \;\Leftarrow\; \begin{cases} \textit{find } z \textit{ such that} \\ \qquad perm(\ell, z) \wedge ord(z)\,. \end{cases}$$

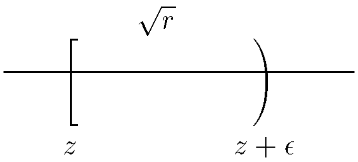
This specification is presented in a background theory of lists of numbers. For a given input object, the list ℓ, the program must return an output object, the list z, satisfying the condition $perm(\ell, z)$, i.e., that z is a permutation of ℓ, and the condition $ord(z)$, i.e., that z is in nondecreasing order. The background theory provides the meaning for the constructs *perm* and *ord*. ∎

Note that the specification provides a clear statement of the purpose of a sorting program, but does not describe how we want the list to be sorted. A sorting program itself, such as *quicksort* or *mergesort*, does describe how the computation is to be performed, but does not state the purpose of the program.

Example (square-root specification). Suppose we want a program to find a rational approximation to the square root of a nonnegative rational; then we may give the specification

$$sqrt(r, \epsilon) \;\Leftarrow\; \begin{cases} \textit{find } z \textit{ such that} \\ \qquad \textit{if } \epsilon > 0 \\ \qquad \textit{then } z^2 \leq r \;\wedge\; r < (z+\epsilon)^2\,. \end{cases}$$

Here we are given the nonnegative rational r and positive rational error tolerance ϵ as inputs. Our desired output z is less than or equal to $\sqrt{r}$, that is, $z^2 \leq r$, but $z + \epsilon$ is strictly greater than $\sqrt{r}$, that is, $r < (z + \epsilon)^2$. In other words, $\sqrt{r}$ lies in the half-open interval $[z, z + \epsilon)$:



Our background theory is that of the nonnegative rationals. ∎

In general, we shall be dealing with specifications of the form

$$f(a) \Leftarrow \textit{find } z \textit{ such that } \mathcal{Q}[a, z],$$

where $\mathcal{Q}[a, z]$ is a sentence of the background theory.

Deductive Software Technologies

Program synthesis is one of several methods to assist in software development that is amenable to deductive techniques. Here we mention some of the other deductive software-development methods, with representative references.

- *Program Verification.* Proving that a given program meets a given specification (Boyer and Moore [BM79]). This is the oldest of the deductive methods.
- *Program Transformation.* Transforming a given program into a more efficient, perhaps less understandable equivalent (Burstall and Darlington [BD77]).
- *Rapid Prototyping.* Assuring a potential user that a specification actually does agree with his expectations (Guttag and Horning [GH80]).
- *Logic Programming.* Executing a program expressed in logic (Kowalski [Kow74]).
- *Debugging.* Exhibiting inputs that cause a program to fail to meet its specification (Shapiro [Sp83]).
- *Modification.* Altering a given program to reflect changes in its specification or environment (Dershowitz [De83]).

In a somewhat different category, we may consider a variety of knowledge-based software development methods (e.g., Smith [Sm85]) which rely on imitating the techniques of the experienced programmer. Automated deduction is exploited here in an auxiliary role; the programming process is not regarded as a task of proving a theorem, but as a task of transformation with many deductive subtasks.

Many researchers in formal methods for software development (e.g., Dijkstra [Dij76]) do regard programming as primarily a deductive process, but are not at all concerned with automating the task; rather, they intend to provide intellectual tools for the programmer.

These methods all rely on deductive techniques, and several of them are less ambitious than full program synthesis. By developing more powerful theorem-proving techniques that are specialized to software-engineering applications, we can make progress in several of these areas at once.

Outline of Deductive Program Synthesis

In this section we give a more detailed outline of program synthesis and its relation to mathematical proofs.

In general, we are given a specification

$$f(a) \Leftarrow \textit{find } z \textit{ such that } \mathcal{Q}[a, z]\,.$$

The theorem corresponding to this specification is

$$(\forall a)(\exists z)\mathcal{Q}[a, z]\,.$$

In other words, for every input a, there exists an output z that satisfies the input-output relation $\mathcal{Q}[a, z]$. The proof is restricted to be sufficiently constructive to indicate a method for finding z in terms of a. That method is expressed by a single term $t[a]$, which can be extracted from the proof. The term indicates which substitutions were made for z to allow the proof to go through. The program we produce is then

$$f(a) \Leftarrow t[a]\,.$$

We describe the method as if there were only one input and output, but in fact we can have several of each. If there is more than one output, we define a separate function for each. In the following example, there are two outputs.

Example (front/last derivation outline). In the theory of finite strings, we would like to construct a program to find, for a given nonempty string s, two outputs: the last character $last(s)$ of s, and the string $front(s)$ of all but the last character of s. For example, if s is the string BADA, $front(s)$ is the string BAD and $last(s)$ is the character A.

The program may be specified as

$$\langle front(s),\ last(s)\rangle \Leftarrow \textit{find } \langle z_1, z_2\rangle \textit{ such that } \left[\begin{array}{l} \textit{if } \neg\,(s = \Lambda) \\ \textit{then } char(z_2) \ \wedge\ s = z_1 * z_2 \end{array}\right].$$

In other words, s is to be decomposed into the concatenation $z_1 * z_2$ of two strings, z_1 and z_2, where z_2 consists of a single character. Here Λ is the empty string. Note that characters are regarded as strings.

The theorem corresponding to the specification is

$$(\forall s)(\exists z_1, z_2)\left[\begin{array}{l} \textit{if } \neg\,(s = \Lambda) \\ \textit{then } char(z_2) \ \wedge\ s = z_1 * z_2 \end{array}\right].$$

The proof is restricted to be sufficiently constructive to indicate a method for finding z_1 and z_2. In this case, the program we shall extract from the proof is

$$front(s) \Leftarrow \begin{cases} \textit{if } char(s) \\ \textit{then } \Lambda \\ \textit{else } head(s) \cdot front\big(tail(s)\big) \end{cases}$$

$$last(s) \Leftarrow \begin{cases} if\ char(s) \\ then\ s \\ else\ last\big(tail(s)\big). \end{cases}$$

Here $head(s)$ and $tail(s)$ are, respectively, the first character and the string of all but the first character of the nonempty string s. Also $char(s)$ is true if s consists of a single character. If c is a character and s is a string, the prefix function $c \cdot s$ yields the result of prefixing c to s. Thus, $c \cdot s$ is the same as $c * s$, but $c \cdot s$ is a basic function defined only for a character and a string. The concatenation function $s_1 * s_2$ is defined in terms of the prefix function, for any two strings s_1 and s_2. ∎

The structure of the proof of the theorem determines the structure of the program we extract. In particular, a case analysis in the proof corresponds to the formation of a conditional or test in the program. The use of the principle of mathematical induction in the proof coincides with the appearance of recursion or other repetitive constructs in the program. If the proof requires some lemmas, the program will invoke some auxiliary subprograms. Of course, different proofs of the theorem may lead to different programs, some of which may be preferable to others.

The phrasing of a specification as a theorem is quite straightforward. If a proof is sufficiently constructive, the extraction of the program is purely mechanical. Thus the main problem of deductive program synthesis is finding a sufficiently constructive proof of the theorem. We now turn our attention to the field of theorem proving, or automated deduction.

Theorem Proving

We may distinguish between decision procedures, which guarantee success at proving theorems within a particular class, and heuristic methods, whose success is not guaranteed. We may also distinguish between automatic systems, which act without human intervention, and interactive systems, which require it.

The theories of interest here, such as those of the nonnegative integers, strings, and trees, are undecidable; no decision procedures exist. We know of no way of restricting the specification or programming language to ensure the successful completion of a proof without also restricting ourselves to a trivial class of specifications and programs. We assume then that our theorem prover will employ heuristic methods or rely on human guidance, probably both.

We have distinguished between automatic and interactive systems, but this distinction is not sharp. Implementers of interactive systems introduce automatic features, to reduce the burden on the user. At the same time, implementers of automatic systems introduce interactive controls, so the user can assist the system to discover proofs that are too difficult to be found automatically.

Although interactive systems are susceptible to gradual automation, most of them are intended to help the user check and flesh out a proof already outlined by hand, rather than to discover a new proof. The logical frameworks embedded in the automatic systems are more conducive to proof discovery.

The emphasis of this paper, however, is on neither the heuristic aspects of theorem proving nor on the design of interactive mechanisms, but rather on the development of a logical framework sufficiently powerful to facilitate the discovery and succinct presentation of nontrivial derivation proofs.

Let us consider some of the theorem-proving systems that have already been developed, to see how appropriate they are for our purpose. We discuss some automatic and interactive systems.

We may classify automatic theorem provers according to the logical theories on which they focus.

- *Predicate Logic with Equality.* Much work has exploited the resolution (J. A. Robinson [Ro65]) and paramodulation (Wos and G. Robinson [WR69]) inference rules for these theories. Theorem provers based on these ideas, such as those developed at the Argonne National Laboratory (McCune [Mc90]), regularly settle open questions in mathematics and logic (Wos and Winker [WW83]), admittedly in areas in which human intuition is weak, such as combinatory logic and equivalential calculus. Recent theorem proving systems for predicate logic with equality have employed term-rewriting systems (Kapur and Narendran [KN84]) and connection methods (Andrews [An81], Bibel [Bi83]), rather than resolution and paramodulation, as the primary inference technique.
- *Theories with Induction.* A separate body of work focuses on proofs requiring the principal of mathematical induction. The Boyer-Moore system [BM79] has been motivated by and applied to large problems in program verification, but has also been applied to the interactive reconstruction of large proofs in mathematics and logic, such as the Gödel Incompleteness theorem (Shankar [Sn85]).

All of this work is relevant to program synthesis, yet it is difficult to find an existing system with all the features we need. We require the ability to prove theorems involving the quantifiers and connectives of first-order logic and the mathematical induction principle. The Argonne systems, for example, do well with pure predicate logic but have no facilities for inductive proofs. The Boyer-Moore system, which specializes in proof by induction, does not prove theorems with existential quantifiers.

Many of the interactive systems have grown out of LCF (Gordon et al [GMW79]), which was based on Scott's Logic of Computable Functions. Although these systems are under user control, they provide the capability to encode commonly repeated patterns of inference as tactics. The system Isabelle (Paulson [Pa89]) arises from LCF but is generic; that is, it allows us to describe a new logic, then prove theorems in that logic (cf. Felty and Miller [FM88]).

Of particular relevance to program synthesis is the development of interactive systems to prove theorems in constructive logics. The Nuprl system (Constable et al [Con86], cf. Coquand and Huet [CH88], Sato [Sa79], Hayashi and Nakano [HN88]) is based on Martin-Löf's constructive logic (Martin-Löf [M-L82], Nordstrom et al [NPS90]) and has been applied to problems in program derivation as well as mathematics.

Although a derivation proof must be sufficiently constructive to allow us to extract a program, it does not need to be carried out in a constructive logic. Typically, most of a derivation proof has no bearing on the program we extract; it deals with showing that a program fragment extracted from some other part of the proof satisfies some additional conditions. Since many intuitively natural steps are not constructive, it is too constraining to carry out the entire derivation proof in a constructive logic. In our treatment, we adopt a classical logic, restricting it to be constructive only when necessary.

Most theorem-proving systems can be adapted to program synthesis and other software-engineering applications. The deductive framework we employ in this paper is a hybrid; it incorporates ideas from resolution and from inductive theorem proving, and it is intended for both interactive and automatic implementation. An interactive synthesis system, based on the theorem prover described in [BMW90], has been implemented.

PRELIMINARIES

In this section we introduce some formal preliminaries. We are a bit brisk here; the section may be skimmed by those familiar with these notions. Those wishing a more detailed explanation may refer to Manna and Waldinger ([MW85], [MW90]).

Language

We first define the *expressions* of our language, which consist of the terms and the sentences.

The *terms* include the constants $a, b, c, \ldots$ and the variables $u, v, w, \ldots$. Terms may be constructed by the repeated application of function symbols $f, g, h, \ldots$ to other terms. For example, $f\big(a,\ g(a,x)\big)$ is a term. Also, if $\mathcal{F}$ is a sentence and s and t are terms, the conditional (*if* $\mathcal{F}$ *then* s *else* t) is a term; we call the *if-then-else* operator a *term constructor.*

Atomic sentences (or *atoms*) are constructed by applying predicate symbols $p, q, r, \ldots$ to terms. For example, $p\big(u,\ f\big(a,\ g(a,x)\big)\big)$ is an atomic sentence. We allow both prefix and infix notations for function and predicate symbols. We include the equality symbol $=$ as a predicate symbol.

Sentences include the truth symbols *true* and *false* and the atomic sentences; they may be constructed by the repeated application of the connectives $\wedge$, $\vee$, $\neg$, ...and the quantifiers $(\forall x)$ and $(\exists x)$ to other sentences. We use the notation *if-then* for implication in place of the conventional arrow or horseshoe. We include a conditional connective *if-then-else*; in other words, if $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{H}$ are sentences then (*if* $\mathcal{F}$ *then* $\mathcal{G}$ *else* $\mathcal{H}$) is also a sentence. We rely on context to distinguish between the conditional connective and the conditional term constructor.

A *closed* expression contains no free (unquantified) variables. A *ground* expression contains no variables at all. A *herbrand* expression is ground and contains neither connectives, term constructors, nor equality symbols; thus $g(a)$ is a herbrand term and $p\big(a,\ f(a,b)\big)$ is a herbrand atom.

Interpretation and Truth

The truth of a sentence is defined only with respect to a particular interpretation. Intuitively speaking, we may think of an interpretation as a situation or case. We adopt the herbrand notion and define an interpretation as a finite or infinite set of herbrand atoms. Informally, we think of the elements of the interpretation as a complete list of the herbrand atoms that are true in the corresponding situation. The truth-value of any closed sentence with respect to the interpretation is determined by the recursive application of the following semantic rules.

- A herbrand atom $\mathcal{P}$ is true under an interpretation $\mathcal{I}$ if $\mathcal{P} \in \mathcal{I}$.
- If a sentence is not closed, we do not define its truth-value. Thus we do not say whether $p(x)$ is true under $\{p(a)\}$. (Henceforth in this section we speak only of closed sentences.)
- A closed sentence $(\mathcal{F} \wedge \mathcal{G})$ is true under $\mathcal{I}$ if $\mathcal{F}$ and $\mathcal{G}$ are both true under $\mathcal{I}$; similarly for the other connectives.
- A closed sentence $(\exists x)\mathcal{F}[x]$ is true under $\mathcal{I}$ if there is a herbrand term t such that $\mathcal{F}[t]$ is true under $\mathcal{I}$; here $\mathcal{F}[t]$ is the result of replacing all free occurrences of x in $\mathcal{F}[x]$ with t. For example, the sentence $(\exists x)p(x)$ is true under the interpretation $\{p(a)\}$ because a is a herbrand term and $p(a)$ is true under $\{p(a)\}$.
- A closed sentence $(\forall x)\exists[x]\mathcal{F}[x]$ is true under $\mathcal{I}$ if, for every herbrand term t, $\mathcal{F}[t]$ is true under $\mathcal{I}$.
- If (*if* $\mathcal{P}$ *then* s *else* t) is a closed term, a closed sentence $\mathcal{F}[$*if* $\mathcal{P}$ *then* s *else* $t]$ is true under $\mathcal{I}$ if the sentence (*if* $\mathcal{P}$ *then* $\mathcal{F}[s]$ *else* $\mathcal{F}[t]$) is true under $\mathcal{I}$.
- For herbrand terms s and t, $s = t$ is true under $\mathcal{I}$ if, for each herbrand atom $\mathcal{P}\langle s\rangle$, $\mathcal{P}\langle s\rangle \in \mathcal{I}$ if and only if $\mathcal{P}\langle t\rangle \in \mathcal{I}$. Here $\mathcal{P}\langle t\rangle$ is obtained from $\mathcal{P}\langle s\rangle$ by replacing exactly one free occurrence of s with t. This holds only when s and t are indistinguishable under $\mathcal{I}$. For example, $a = b$ is true under the interpretation $\{p(a), p(b)\}$, but false under the interpretation $\{q(a, b),\ q(a, a),\ q(b, b)\}$; $q(a, a)$ belongs to the latter interpretation, but $q(b, a)$ does not. In general, if a closed sentence $s = t$ is true under $\mathcal{I}$, we shall also say that $s = t$ under $\mathcal{I}$ or that s and t are *equal* under $\mathcal{I}$.

Henceforth we will often say "sentence" when we mean "closed sentence."

Models and Theories

An interpretation $\mathcal{I}$ is a *model* for a finite or infinite set of (closed) sentences $\mathcal{S}$ if every sentence in $\mathcal{S}$ is true under $\mathcal{I}$. Thus the interpretations $\{p(a)\}$ and $\{p(b)\}$ are models for the set of sentences $\{(\exists x)p(x),\ p(a) \vee p(b)\}$ but the interpretation $\{p(b)\}$ is not a model for the set of sentences $\{p(a)\}$.

A set of sentences $\mathcal{S}$ implies a sentence $\mathcal{F}$ if $\mathcal{F}$ is true under every model for $\mathcal{S}$. For example, the set $\{p(a)\}$ implies the sentence $(\exists x)p(x)$. The theory TH defined by a set of sentences $\mathcal{S}$ is the set of all closed sentences implied by $\mathcal{S}$; this is also called the *deductive closure* of $\mathcal{S}$. We say that the sentences belonging to TH are *valid* in the theory. We call $\mathcal{S}$ the set of *axioms* for the theory TH.

The valid sentences of a theory are true under every model for the theory. The *contradictory* sentences of the theory are defined to be those that are false under every model for the theory. A sentence $\mathcal{F}$ is contradictory in the theory if and only if its negation $\neg\mathcal{F}$ is valid in the theory.

The theory defined by the empty set $\{\ \}$ of axioms is *predicate logic*, PL. For example, $(\exists x)p(x) \vee (\forall x)\neg p(x)$ is a valid sentence of predicate logic. Any interpretation is a model for predicate logic.

The *total reflexive* theory TR is defined by the following two axioms.

$$(\forall u)[u \succeq u] \qquad (reflexivity)$$

$$(\forall u)(\forall v)[u \succeq v \;\vee\; v \succeq u] \quad (totality)$$

By convention, we omit outermost universal quantifiers from axioms. Thus, we may write the axioms for the total reflexive theory TR as

$$u \succeq u \qquad (reflexivity)$$

$$u \succeq v \;\vee\; v \succeq u \quad (totality)$$

The sentence

$$(\forall x)(\forall y)(\exists z)[z \succeq x \;\wedge\; z \succeq y]$$

is valid in this theory. ∎

When we say that a (closed) sentence is valid, without specifying a theory, we mean that it is valid in predicate logic. If a sentence is valid (in predicate logic), it is valid in any theory.

The models for a theory are the same as the models for its axioms. Intuitively speaking, a model for a theory corresponds to a situation that could possibly happen. For example, an interpretation that contains neither $a \succeq b$ nor $b \succeq a$ is not a model for the total reflexive theory TR, because it violates the totality axiom.

Substitutions

A *substitution* is a set $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ of replacement pairs $x_i \leftarrow t_i$, where the x_i are distinct variables, the t_i are terms, and each x_i is distinct from its corresponding t_i. Thus, $\{x \leftarrow y,\, y \leftarrow g(x)\}$ is a substitution but $\{x \leftarrow a,\, x \leftarrow b\}$ and $\{x \leftarrow x\}$ are not. The *empty* substitution $\{\,\}$ contains no replacement pairs.

If e is an expression and $\theta : \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ is a substitution, then $e\theta$, the *result of applying θ to e*, is obtained by *safely* replacing each free occurrence of x_i in e with the corresponding term t_i. (The *safety* condition requires that certain quantified variables y in e be given a new name y' if some of the terms t_i also contain occurrences of y. For details, see Manna and Waldinger [MW85].) Applying the empty substitution leaves an expression unchanged; that is, $e\{\,\} = e$ for all expressions e. We say that any expression $e\theta$ is an *instance* of e.

The *composition* $\theta\lambda$ of two substitutions θ and λ is a substitution with the property that $e(\theta\lambda) = (e\theta)\lambda$ for all expressions e. For example,

$$\begin{aligned}
\{x \leftarrow y\}\{y \leftarrow a\} &= \{x \leftarrow a, y \leftarrow a\}\\
\{x \leftarrow y\}\{y \leftarrow x\} &= \{y \leftarrow x\}\\
\{x \leftarrow y\}\{x \leftarrow a\} &= \{x \leftarrow y\}\,.
\end{aligned}$$

Composition is associative but not commutative. The empty substitution is an identity under composition.

A substitution is a permutation if the terms t_i are the same as the variables x_i, in some order. Thus $\{x \leftarrow y,\ y \leftarrow z,\ z \leftarrow x\}$ is a permutation; $\{x \leftarrow y\}$ is not. Permutations are the substitutions with inverses: That is, π is a permutation if and only if there is some substitution π^{-1} such that $\pi\pi^{-1} = \{\,\}$.

A substitution θ is *more general* than a substitution ϕ if there exists a substitution λ such that

$$\theta\lambda = \phi\,.$$

For example, $\{x \leftarrow y\}$ is more general than $\{x \leftarrow a,\ y \leftarrow a\}$, because $\{x \leftarrow a,\ y \leftarrow a\} = \{x \leftarrow y\}\{y \leftarrow a\}$. It follows that any substitution θ is more general than itself and the empty substitution $\{\,\}$ is more general than any substitution θ.

A substitution θ is a *unifier* of two expressions d and e if

$$d\theta = e\theta\,.$$

For example, $\{x \leftarrow a,\ y \leftarrow b\}$ is a unifier of the two expressions $p(x, b)$ and $p(a, y)$. If two expressions have a unifier, they are said to be *unifiable*.

A unifier of d and e is *most-general* if it is more general than any unifier of d and e. For example, $\{x \leftarrow y\}$ and $\{y \leftarrow x\}$ are most-general unifiers of x and y. The substitution $\theta : \{x \leftarrow a,\ y \leftarrow a\}$ is a unifier of x and y, and both $\{x \leftarrow y\}$ and $\{y \leftarrow x\}$ are more general than θ.

A *unification algorithm* is a procedure for testing whether two expressions are unifiable. If so, it returns a most-general unifier; otherwise, it returns a special object, *nil*, say, which is distinct from any substitution.

THE DEDUCTIVE TABLEAU

Our proofs are represented by a two-dimensional structure, the *deductive tableau*. Each row in a tableau contains a sentence, either an *assertion* or a *goal*, and an optional term, the output entry. In general, in a given row, there may be one output entry for each output of the desired program. Thus, typical rows in a tableau have the following forms.

assertions	goals	$f_1(a)$	...	$f_n(a)$
$\mathcal{A}_1$		s_1		s_n
	$\mathcal{G}_1$	t_1		t_n

rows

output columns

The proof itself is represented by the assertions and goals of the tableau; the output entries serve for extracting a program from the proof. Usually we speak as if our tableaux have only a single output column, but in fact the results apply when there are several output columns as well.

Before we describe the meaning of a tableau, let us look at an example.

Example (deductive tableau).

assertions	goals	$ub(a_1, a_2)$
	$z \succeq a_1 \ \wedge \ z \succeq a_2$	z
$u \succeq u$		
	$a_1 \succeq a_2$	a_1
	true	*if* $a_1 \succeq a_2$ *then* a_1 *else* a_2

This tableau is part of the derivation of a program to find an upper bound for two objects a_1 and a_2 in the total reflexive theory TR. ∎

Suiting a Tableau

We have said that a tableau may represent a proof and a derivation; it may also be regarded as a specification. Specifications describe sets of permissible output objects, which are identified with ground terms. In this section, we gradually define what it means for a ground term to satisfy a tableau.

We first restrict our attention to a particular interpretation and a single row of a tableau.

Definition (suiting a row). A closed term t *suits* a row $\boxed{\mathcal{A}}\boxed{\phantom{\mathcal{G}}}\boxed{s}$ $\left(\text{or, respectively, } \boxed{\phantom{\mathcal{A}}}\boxed{\mathcal{G}}\boxed{s}\right)$ under an interpretation $\mathcal{I}$ if, for some substitution λ, the following two conditions are satisfied:

- *Truth condition.* The sentence $\mathcal{A}\lambda$ is closed and false under $\mathcal{I}$ (or, respectively, the sentence $\mathcal{G}\lambda$ is closed and true under $\mathcal{I}$).

- *Output condition.* If there is an output entry s, the term $s\lambda$ is closed and $s\lambda$ equals t under $\mathcal{I}$.

In case the output entry s is absent, the output condition holds vacuously. We call λ a *suiting* substitution. ∎

Example (suiting a row). If $a_1 \succeq a_2$ is true under an interpretation $\mathcal{I}$, the term a_1 suits the row

	$z \succeq a_2$	z

under $\mathcal{I}$. To see this, we take the suiting substitution λ to be $\{z \leftarrow a_1\}$. The truth condition holds because $(z \succeq a_2)\lambda$, that is, $a_1 \succeq a_2$, is closed and true under $\mathcal{I}$. The output condition holds because $z\lambda$, that is, a_1, is closed and equal to a_1 under $\mathcal{I}$.

In this example, the term a_1 is actually identical to the instance $z\lambda$ of the output entry z. The conditional term (*if* $a_1 \succeq a_2$ *then* a_1 *else* a_2) is also equal to this instance of z under $\mathcal{I}$, because $a_1 \succeq a_2$ is true, but the two terms are not identical. In fact, the conditional term (*if* $a_1 \succeq a_2$ *then* a_1 *else* a_2) also suits this row under $\mathcal{I}$. ∎

If a row has no output entry, the output condition for suiting a row always holds. This means that, under an interpretation, if some closed term suits the row, then any closed term suits the row, since the truth condition does not depend on the term. In a sense, a missing output entry may be thought of as a "don't care" condition.

We have defined what it means to suit a single row; now we say what it means to suit an entire tableau.

Definition (suiting a tableau). Under an interpretation, a closed term *suits* a tableau if it suits some row of the tableau. ∎

If we think of the tableau as a specification, and the interpretation as a situation or case, the closed terms that suit the tableau coincide with the outputs that will meet the specification in that case.

Example (suiting a tableau). Let $\mathcal{T}$ be the following tableau:

	$a_1 \succeq a_2$	a_1
	$\neg(a_1 \succeq a_2)$	a_2

If $a_1 \succeq a_2$ is true under $\mathcal{I}$, then a_1 suits $\mathcal{T}$ under $\mathcal{I}$, with the empty suiting substitution $\{\,\}$. If, on the other hand, $\neg(a_1 \succeq a_2)$ is true under $\mathcal{I}$, then a_2 suits $\mathcal{T}$ under $\mathcal{I}$. In either case, the conditional term (*if* $a_1 \succeq a_2$ *then* a_1 *else* a_2) suits $\mathcal{T}$ under $\mathcal{I}$. ∎

Satisfying a Tableau

The notion of suiting a tableau depends on the interpretation; a term may suit a tableau under one interpretation and not under another. In that sense suiting is analogous to truth for a sentence. We now introduce a notion of "satisfying" a tableau, which is independent of the particular interpretation. That notion is analogous to validity for a sentence.

Definition (satisfying a tableau). In a theory TH, a closed term t *satisfies* a tableau $\mathcal{T}$ if t suits $\mathcal{T}$ under every model of TH. ∎

If we think of the tableau as a specification, t corresponds to a program that satisfies the specification.

Example (satisfying a tableau). Suppose $\mathcal{T}$ is the following tableau:

assertions	goals	$f(a_1, a_2)$
	$z \succeq a_2$	z
	$a_2 \succeq a_1$	a_2

Let our background theory be the total reflexive theory TR. Then the closed term

t: *if* $a_1 \succeq a_2$ *then* a_1 *else* a_2

satisfies $\mathcal{T}$ in TR.

To see this, consider an arbitrary model $\mathcal{I}$ for TR. We distinguish between two cases:

Case: $a_1 \succeq a_2$ is true under $\mathcal{I}$

In this case, t equals a_1 under $\mathcal{I}$. Then, t suits the first row

	$z \succeq a_2$	z

under $\mathcal{I}$, as we have seen. Therefore, t suits $\mathcal{T}$ under $\mathcal{I}$.

Case: $a_1 \succeq a_2$ is false under $\mathcal{I}$

In this case, t equals a_2 under $\mathcal{I}$. Also (by the totality axiom, since $\mathcal{I}$ is a model for the total reflexive theory TR), $a_2 \succeq a_1$ is true under $\mathcal{I}$. Thus, t suits the second row

	$a_2 \succeq a_1$	a_2

under $\mathcal{I}$. Therefore, t suits $\mathcal{T}$ under $\mathcal{I}$.

Thus, for any model $\mathcal{I}$ for the theory TR, t suits $\mathcal{T}$ under $\mathcal{I}$. Hence, t satisfies the tableau in TR. ∎

Equivalence Between Tableaux

We introduce two distinct relations of similarity between tableaux. The stronger relation, equivalence, requires that the two tableaux always have the same suiting terms.

Definition (equivalence of tableaux). Two tableaux $\mathcal{T}_1$ and $\mathcal{T}_2$ are *equivalent* in the theory TH, written $\mathcal{T}_1 \leftrightarrow \mathcal{T}_2$, if and only if for every closed term t and every model $\mathcal{I}$ for TH,

t suits $\mathcal{T}_1$ under $\mathcal{I}$
if and only if
t suits $\mathcal{T}_2$ under $\mathcal{I}$. ∎

That is, for $\mathcal{T}_1$ and $\mathcal{T}_2$ to be equivalent in TH, they must have the same suiting terms under each model for the theory. When we say that two tableaux are equivalent without specifying a theory, we mean that they are equivalent in predicate logic. If two tableaux are equivalent (in predicate logic), they are equivalent in any theory.

Examples of equivalent tableaux will be provided by the following basic properties. The proof of one of these properties is provided; the others are similar.

Property (duality). For any sentences $\mathcal{A}$ and $\mathcal{G}$ and optional term s, we have

$$\begin{array}{|c|c||c|} \hline \quad\mathcal{A}\quad & \quad\quad & \quad s \quad \\ \hline \end{array} \quad\longleftrightarrow\quad \begin{array}{|c|c||c|} \hline \quad\quad & \quad\neg\mathcal{A}\quad & \quad s \quad \\ \hline \end{array}$$

$$\begin{array}{|c|c||c|} \hline \quad\quad & \quad\mathcal{G}\quad & \quad s \quad \\ \hline \end{array} \quad\longleftrightarrow\quad \begin{array}{|c|c||c|} \hline \quad\neg\mathcal{G}\quad & \quad\quad & \quad s \quad \\ \hline \end{array} \quad \blacksquare$$

In other words, any assertion $\mathcal{A}$ is equivalent to a goal $(\neg\mathcal{A})$, with the same output entry s, if any. Any goal $\mathcal{G}$ is equivalent to an assertion $(\neg\mathcal{G})$, also with the same output entry.

It will follow that, for any tableau, we can push all the assertions into the goal column, or all the goals into the assertion column, by negating them, thereby obtaining an equivalent tableau. Nevertheless, the distinction between assertions and goals has intuitive appeal and possible strategic power, so we retain it.

The equivalence relation between tableaux has the *substitutivity* property that if we replace any subtableau of a given tableau with an equivalent tableau, we obtain an equivalent tableau. Hence, the duality property allows us to push any assertion of a tableau into the goal column by negating it, obtaining an equivalent tableau.

Property (renaming). For any sentences $\mathcal{A}$ and $\mathcal{G}$, optional term s, and permutation π, we have

$$\begin{array}{|c|c||c|} \hline \quad\mathcal{A}\quad & \quad\quad & \quad s \quad \\ \hline \end{array} \quad\longleftrightarrow\quad \begin{array}{|c|c||c|} \hline \quad\mathcal{A}\pi\quad & \quad\quad & \quad s\pi \quad \\ \hline \end{array}$$

$$\begin{array}{|c|c||c|} \hline \quad\quad & \quad\mathcal{G}\quad & \quad s \quad \\ \hline \end{array} \quad\longleftrightarrow\quad \begin{array}{|c|c||c|} \hline \quad\quad & \quad\mathcal{G}\pi\quad & \quad s\pi \quad \\ \hline \end{array} \quad \blacksquare$$

Applying a permutation to a row has the effect of systematically renaming its free variables. For example, by applying the permutation $\pi : \{x \leftarrow y,\ y \leftarrow z,\ z \leftarrow x\}$ to the assertion

$$\begin{array}{|c|c||c|} \hline \quad p(x,y)\quad & \quad\quad\quad & \quad f(x) \quad \\ \hline \end{array}$$

we obtain the assertion

$$\begin{array}{|c|c||c|} \hline \quad p(y,z)\quad & \quad\quad\quad & \quad f(y) \quad \\ \hline \end{array}$$

The property tells us that these two rows are equivalent.

The renaming property tells us that we can systematically rename the free variables of any row, obtaining an equivalent tableau.

We prove the renaming property for a goal row.

Proof (renaming property). Suppose the closed term t suits the row

	$\mathcal{G}$	s

under interpretation $\mathcal{I}$, with suiting substitution λ. Then, by the truth condition,

$(*)$ $\mathcal{G}\lambda$ is closed and true under $\mathcal{I}$

and, by the output condition,

$(\dagger)$ $s\lambda$ is closed and equal to t under $\mathcal{I}$.

We show that then t also suits the row

	$\mathcal{G}\pi$	$s\pi$

with suiting substitution $\pi^{-1}\lambda$, where π^{-1} is the inverse of the permutation π.

To show this, we show the truth condition,

$(\mathcal{G}\pi)(\pi^{-1}\lambda)$ is closed and true under $\mathcal{I}$,

and the output condition

$(s\pi)(\pi^{-1}\lambda)$ is closed and equal to t under $\mathcal{I}$.

But these follow from the conditions $(*)$ and $(\dagger)$, because, by properties of substitutions, $(\mathcal{G}\pi)(\pi^{-1}\lambda) = \mathcal{G}(\pi\pi^{-1})\lambda = \mathcal{G}\{\,\}\lambda = \mathcal{G}\lambda$, and similarly for s.

In the other direction, we assume that t suits the row

	$\mathcal{G}\pi$	$s\pi$

with suiting substitution λ, and can show that t also suits the original row

	$\mathcal{G}$	s

with suiting substitution $\pi\lambda$. ∎

Property (instance). For any sentences $\mathcal{A}$ and $\mathcal{G}$, optional term s, and substitution θ, we have

$$\begin{array}{|c|c||c|}\hline \mathcal{A} & \quad & s \\ \hline \end{array} \quad \longleftrightarrow \quad \begin{array}{|c|c||c|}\hline \mathcal{A} & \quad & s \\ \hline \mathcal{A}\theta & & s\theta \\ \hline \end{array}$$

$$\begin{array}{|c|c||c|}\hline \quad & \mathcal{G} & s \\ \hline \end{array} \quad \longleftrightarrow \quad \begin{array}{|c|c||c|}\hline \quad & \mathcal{G} & s \\ \hline & \mathcal{G}\theta & s\theta \\ \hline \end{array} \quad ∎$$

It follows that we may add to a tableau any instance of any of its rows, obtaining an equivalent tableau. Note that, while the duality and renaming properties allow us to replace one row with another, the instance property requires that we retain the old row while adding the new one. If we replaced the row, we would not necessarily retain equivalence.

The following property allows us to add or remove from a tableau any valid assertion, and retain the tableau's equivalence. We restrict our attention to a fixed theory TH.

Property (valid assertion and contradictory goal). Suppose $\mathcal{A}$ is a sentence where every ground instance $\mathcal{A}\theta$ is valid in theory TH; suppose $\mathcal{G}$ is a sentence whose every ground instance $\mathcal{G}\theta$ is contradictory in TH. Then for any tableau $\mathcal{T}$ and term s,

$$\mathcal{T} \quad \longleftrightarrow \quad \begin{array}{|c|c||c|}\hline \multicolumn{3}{|c|}{\mathcal{T}} \\ \hline \mathcal{A} & & s \\ \hline \end{array}$$

$$\mathcal{T} \quad \longleftrightarrow \quad \begin{array}{|c|c||c|}\hline \multicolumn{3}{|c|}{\mathcal{T}} \\ \hline & \mathcal{G} & s \\ \hline \end{array}$$

in theory TH. In other words, $\mathcal{A}$ may be added as an assertion, or $\mathcal{G}$ as a goal, to any tableau, yielding an equivalent tableau. ∎

It follows from the valid assertions property that any row $\begin{array}{|c|c||c|}\hline \textit{true} & & s \\ \hline \end{array}$ or $\begin{array}{|c|c||c|}\hline & \textit{false} & s \\ \hline \end{array}$ can be dropped from any tableau. These are sometimes called *trivial* rows.

We have defined validity in a theory for closed sentences only. However, if $\mathcal{A}$ is an assertion in a tableau that is not closed, we often say that $\mathcal{A}$ is a valid sentence when we really mean that every closed instance of $\mathcal{A}$ is valid. The valid assertion property can then be paraphrased to say that a valid assertion can be added to any tableau, preserving its equivalence.

The following property tells us more about what it means when a row lacks an output entry.

Property (no output). A row (assertion or goal) with no output entry is equivalent to one whose output entry is a new variable, that is, a variable that does not occur free in the row.

$$\mathcal{T}: \begin{array}{|c||c|}\hline \qquad\qquad\qquad & \qquad\qquad \\ \hline \end{array} \quad \longleftrightarrow \quad \mathcal{T}_u: \begin{array}{|c||c|}\hline \qquad\qquad\qquad & u \\ \hline \end{array} \quad ∎$$

The rationale here is that if some closed term suits either of these rows, then any closed term will. More precisely, a closed term t suits $\mathcal{T}$ with suiting substitution $\{x_1 \leftarrow t_1, \ \ldots, \ x_n \leftarrow t_n\}$ if and only if t suits $\mathcal{T}_u$ with suiting substitution $\{u \leftarrow t, \ x_1 \leftarrow t_1, \ \ldots, \ x_n \leftarrow t_n\}$.

Primitive Expressions

For some purposes, the notion of equivalence is too strong. We may not care if two tableaux are *suited* by the same closed terms, for each model for the theory, so long as they are *satisfied* by the same closed terms. And we may not care if they are satisfied by precisely the same closed terms, so long as they are satisfied by the same closed terms that correspond to computer programs, that is, those that we know how to compute. This latter idea is captured in the notion of primitive terms.

Definition (primitive expression). Assume we are given a finite set of constant, function, and predicate symbols, called the *primitive set.* An expression is said to be *primitive* if

- It is quantifier-free.
- All of its constant, function, and predicate symbols belong to the primitive set. ∎

Note that a primitive expression may contain variables.

Intuitively speaking, the primitive expressions are those we know how to compute, in terms of the variables and the elements of the primitive set. Typically, the primitive set will include the basic operators of the theory plus those function symbols for which we have already derived programs. For example, in deriving a program to compute the multiplication function in the theory of the nonnegative integers, we typically include the constant symbol 0, the addition function symbol $+$, and the equality predicate symbol $=$ in the primitive set.

We can now define a relation of similarity, weaker than equivalence, between tableaux.

Definition (primitively similar). Two tableaux are *primitively similar* in theory TH if they have the same primitive satisfying terms, that is, for every closed primitive term t,

t satisfies $\mathcal{T}_1$ in TH
if and only if
t satisfies $\mathcal{T}_2$ in TH. ∎

Evidently, if two tableaux are equivalent they are primitively similar. Let us give an example to show that primitive similarity is a strictly weaker notion than equivalence.

Example (equivalence versus primitive similarity). Consider the two tableaux

$\mathcal{T}_p$:

	$p(a)$	a

$\mathcal{T}_q$:

	$q(a)$	a

These tableau are not equivalent. If $\mathcal{I}_p$ is the interpretation $\{p(a)\}$, a suits $\mathcal{T}_p$ under $\mathcal{I}_p$ but a does not suit $\mathcal{T}_q$ under $\mathcal{I}_p$.

On the other hand, in the theory of predicate logic, no closed term satisfies $\mathcal{T}_p$; in particular, no term suits $\mathcal{T}_p$ under the empty interpretation $\{\,\}$, because $p(a)$ is false under $\{\,\}$. Similarly, no closed term suits $\mathcal{T}_q$ in predicate logic either. Hence, the two tableaux are primitively similar, because they are satisfied by precisely the same primitive satisfying terms, namely, none. ∎

If two tableaux are primitively similar, they specify the same class of programs.

PROPERTIES OF DEDUCTION RULES

Deduction rules add new rows to a tableau. They do not necessarily preserve equivalence, but they do preserve primitive similarity, that is, they maintain the set of primitive closed terms that satisfy the tableau. Thus the program specified by the tableau is unchanged by the application of deduction rules.

Definition (soundness). A rule for adding new rows to a tableau is *sound* in theory TH if the same primitive closed terms satisfy the tableau in TH before and after applying the rule. ∎

We shall guarantee that each of our deduction rules is sound in the background theory.

Let us introduce some terminology for speaking about deduction rules. We use the following notation to describe a rule:

$\mathcal{T}_r$	$\mathcal{A}_r$		
		$\mathcal{G}_r$	
$\mathcal{T}_g$	$\mathcal{A}_g$		
		$\mathcal{G}_g$	

Here the assertions $\mathcal{A}_r$ and the goals $\mathcal{G}_r$ are the *required* rows $\mathcal{T}_r$, which must be present in the tableau if the rule is to be applied. The assertions $\mathcal{A}_g$ and the goals $\mathcal{G}_g$ are the *generated* rows $\mathcal{T}_g$, which may be added to the tableau by the rule.

The *old* tableau refers to the tableau before the application of deduction rules; if the rule is applicable, the required rows form a subtableau $\mathcal{T}_r$ of the old tableau. The *new* tableau refers to the tableau after application of the rule; it is the union of the old tableau and the generated tableau $\mathcal{T}_g$.

Although we are not yet ready to introduce the deduction rules of our system, we mention one of them as an illustration.

Example (if-split rule). In tableau notation, the if-split rule is written

	if $\mathcal{A}$ *then* $\mathcal{G}$	s
$\mathcal{A}$		s
	$\mathcal{G}$	s

In other words, if a row of form (*if* $\mathcal{A}$ *then* $\mathcal{G}$) is present in the tableau, then we may add the new assertion $\mathcal{A}$ and the new goal $\mathcal{G}$. The output entry s for the required goal (*if* $\mathcal{A}$ *then* $\mathcal{G}$), if any, is inherited by the generated assertion $\mathcal{A}$ and the generated goal $\mathcal{G}$. ∎

Description of the Derivation Process

At this point we describe the derivation process and relate it to the deductive-tableau notation.

We are given a specification

$$f(a) \;\Leftarrow\; \textit{find } z \textit{ such that } \mathcal{Q}[a,z]$$

in theory TH. We assume that z is the only free variable in $\mathcal{Q}[a,z]$. We are also given a set of primitive symbols; to allow the formation of recursive programs, we include f in the primitive set.

We form the *initial* tableau

assertions	goals	$f(a)$
	$\mathcal{Q}[a,z]$	z
$\mathcal{A}_1$		
$\vdots$		
$\mathcal{A}_n$		

The input a is taken to be a constant; the output z is a variable. The assertions $\mathcal{A}_1, \ldots, \mathcal{A}_n$ in the initial tableaux are known to be valid in TH.

The deductive process proceeds by the application of sound deduction rules to the tableau, which add new rows while maintaining primitive similarity.

The process continues until we obtain a *final* row, either the assertion

false		t

or the goal

	true	t

where t is a ground primitive term. At this point we may stop the derivation process. The program we obtain is

$$f(a) \Leftarrow t\,.$$

Example (derivation process). In the theory of finite strings, we want to derive a program that, for a given nonempty string s, returns the last character of s and the string of all but the last character of s. Our specification is

$$\langle \textit{front}(s), \textit{last}(s)\rangle \;\Leftarrow\; \begin{cases} \textit{find } \;\langle z_1, z_2\rangle \textit{ such that} \\ \qquad \textit{if } \neg\,(s = \Lambda) \\ \qquad \textit{then } \textit{char}(z_2) \;\wedge\; s \;=\; z_1 * z_2\,. \end{cases}$$

In other words, we want to decompose s into the concatenation $z_1 * z_2$ of a string z_1 and a character z_2; then z_2 is the last character of s and z_1 is the string of all but z_2. We assume s is not equal to the empty string Λ. Note that for this program there are two outputs, z_1 and z_2. That is, we need to compute two functions, *front* and *last*. The primitive set includes all the basic constant, function, and predicate symbols of the theory of strings, as well as the function symbols *front* and *last*.

The corresponding initial tableau then contains the goal

assertions	goals	$front(s)$	$last(s)$
	$if\ \neg(s = \Lambda)$ $then\ char(z_2)\ \wedge\ s = z_1 * z_2$	z_1	z_2

Here the input s is a constant and the outputs z_1 and z_2 are variables. Properties of the theory of strings are also included in the initial tableau as assertions. For instance, the axioms for the concatenation function are represented as the assertions

$\Lambda * y = y$			
$if\ char(u)$ $then\ (u \cdot y_1) * y_2\ =\ u \cdot (y_1 * y_2)$			

By the application of deduction rules, new rows are added to the tableau, obtaining a primitively similar tableau. The process continues until ultimately we obtain the final goal:

	true	$if\ char(s)$ $then\ \Lambda$ $else\ head(s) \cdot$ $front\big(tail(s)\big)$	$if\ char(s)$ $then\ s$ $else\ last\big(tail(s)\big)$

The program we extract from the proof is then

$$front(s) \quad \Leftarrow \quad \begin{cases} if\ char(s) \\ then\ \Lambda \\ else\ head(s) \cdot front\big(tail(s)\big) \end{cases}$$

$$last(s) \quad \Leftarrow \quad \begin{cases} if\ char(s) \\ then\ s \\ else\ last\big(tail(s)\big). \end{cases} \quad \blacksquare$$

The correctness of the derivation process depends on two properties of tableaux. We begin with a definition.

Definition (correctness). A program $f(a) \Leftarrow t[a]$ is *correct with respect to* the specification

$$f(a) \;\Leftarrow\; \textit{find } z \textit{ such that } \mathcal{Q}[a,z]$$

if the sentence

$$(\forall x)\, \mathcal{Q}[x, f(x)]$$

is valid in the theory TH augmented with the additional axiom

$$(\forall x)\big[f(x) = t[x]\big]\,. \quad \blacksquare$$

The additional axiom we add to the theory is the definition of the new program f.

Property (initial tableau). If any closed term $t[a]$ satisfies the initial tableau

assertions	goals	$f(a)$
	$\mathcal{Q}[a,z]$	z
$\mathcal{A}_1$		
$\vdots$		
$\mathcal{A}_n$		

in theory TH, the program $f(a) \Leftarrow t[a]$ is correct with respect to the specification

$$f(a) \;\Leftarrow\; \textit{find } z \textit{ such that } \mathcal{Q}[a,z]$$

in TH. $\blacksquare$

The proof is omitted.

Property (final tableau). A closed term t satisfies any tableau containing the final assertion

false		t

or the final goal

	true	t

in any theory TH. $\blacksquare$

Let us prove this.

Proof (final tableau property). Suppose the tableau contains the final goal

	true	t

Then for any model $\mathcal{I}$ of the theory, the truth condition holds because *true* is true under $\mathcal{I}$, and the output condition holds because t equals t under $\mathcal{I}$. $\blacksquare$

Justification of a Deduction Rule

Our deductive system will have several deduction rules. Furthermore, if we wish to apply the system to a particular theory, it may be convenient to introduce new rules peculiar to that theory. To establish the soundness of these rules, we introduce a general method for justifying deduction rules.

For each rule, we formulate a justification condition. If the justification condition holds, then the rule is sound. This is the content of the following result.

Property (justification). A deduction rule is sound in theory TH if the following *justification* condition holds:

> for any model $\mathcal{I}_r$ for theory TH,
> there exists a model $\mathcal{I}_g$ for TH such that
>   for any closed primitive term t,
>   if t suits the generated tableau $\mathcal{T}_g$ under $\mathcal{I}_g$
>   then t suits the required tableau $\mathcal{T}_r$ under $\mathcal{I}_r$
>   and
> t suits the old tableau $\mathcal{T}_o$ under $\mathcal{I}_g$
>   if and only if
> t suits $\mathcal{T}_o$ under $\mathcal{I}_r$. ∎

The justification condition is exactly what we need to establish that when we add the generated rows to the tableau we are not altering the set of primitive satisfying terms.

Proof (justification property). Suppose that the justification condition holds for a deduction rule. We would like to show that the rule is sound. In other words, we must show that the new tableau and the old tableau specify the same class of primitive closed terms. Because we are adding new rows but not deleting any, we cannot lose any primitive closed terms in applying the rule; we merely must ensure that we do not gain any. In other words, we must guarantee that for any primitive closed term t,

> if t satisfies $\mathcal{T}_n$ in theory TH
> then t satisfies $\mathcal{T}_o$ in TH.

Suppose t does satisfy $\mathcal{T}_n$ in TH; we must show that t also satisfies $\mathcal{T}_o$. Consider an arbitrary model $\mathcal{I}_r$ for TH; we would like to show that

> t suits $\mathcal{T}_o$ under $\mathcal{I}_r$

We have supposed that the justification condition holds for this deduction rule. Let $\mathcal{I}_g$ be the model corresponding to $\mathcal{I}_r$ whose existence is guaranteed by the justification condition. Because we have supposed that t satisfies $\mathcal{T}_n$ in theory TH, we know that

> t suits $\mathcal{T}_n$ under $\mathcal{I}_g$.

The new tableau $\mathcal{T}_n$ consists of the original rows $\mathcal{T}_o$ plus the generated rows $\mathcal{T}_g$. To suit the entire tableau $\mathcal{T}_n$, the term must suit one of these two subtableaux. We distinguish between two cases.

Case: t suits $\mathcal{T}_o$ under $\mathcal{I}_g$.

Then, by the justification condition,

t suits $\mathcal{T}_o$ under $\mathcal{I}_r$,

as we wanted to show.

Case: t suits $\mathcal{T}_g$ under $\mathcal{I}_g$.

Then, by the justification condition again,

t suits $\mathcal{T}_r$ under $\mathcal{I}_r$.

But since $\mathcal{T}_r$ is a subtableau of the old tableau $\mathcal{T}_o$, we have

t suits $\mathcal{T}_o$ under $\mathcal{I}_r$,

as we wanted to show. ∎

The justification property can be used to show soundness of rules that do not preserve the equivalence of the tableau. If a rule does preserve equivalence, it is automatically sound, and there is a simpler way to show that it preserves equivalence.

Property (justification for equivalence). A deduction rule preserves equivalence in theory TH if the following justification condition for equivalence holds:

for any model $\mathcal{I}$ for theory TH,
for any closed term t,
if t suits the generated tableau $\mathcal{T}_g$ under $\mathcal{I}$
then t suits the required tableau $\mathcal{T}_r$ under $\mathcal{I}$. ∎

Proof (justification for equivalence property). Suppose that the justification condition for equivalence holds for a deduction rule. We would like to show that the rule preserves equivalence. In other words, for each model $\mathcal{I}$ for TH, we must show that the sets of terms that suit the two tableaux are the same. Because the rule adds but never deletes rows, we cannot lose any suiting terms in applying the rule; but, we must ensure that we do not gain any. In other words, we must show that, for any closed term t,

if t suits $\mathcal{T}_n$ under $\mathcal{I}$
then t suits $\mathcal{T}_0$ under $\mathcal{I}$.

Suppose t does suit $\mathcal{T}_n$ under $\mathcal{I}$; we must show that t also suits $\mathcal{T}_0$. Because t suits $\mathcal{T}_n$, it must suit either the original subtableau $\mathcal{T}_0$ (as we wanted to show) or the generated subtableau $\mathcal{T}_g$ under $\mathcal{I}$.

If t suits $\mathcal{T}_g$ under $\mathcal{I}$, the justification condition for equivalence tells us that it also suits $\mathcal{T}_r$, and therefore $\mathcal{T}_0$, under $\mathcal{I}$, as we wanted to show. ∎

Let us use the justification condition for equivalence to show the soundness of the if-split rule.

Property (soundness of if-split). The if-split rule preserves equivalence of tableaux, and hence is sound, in any theory TH. ▮

The proof of the soundness of the if-split rule requires a technical notion that will also be useful later.

Definition (closing substitution). Let e be any expression, $y_1, \ldots, y_n$ be a complete list of all the free variables in e, and a be a constant. Then the substitution $\lambda_a = \{y_1 \leftarrow a, \ldots, y_n \leftarrow a\}$ is a *closing substitution* for e.

In the case in which there are no free variables in e, that is, if e is closed, we take the closing substitution $\lambda_a = \{\ \}$. ▮

Note that, if λ_a is a closing substitution for e, then $e\lambda_a$ is closed.

Proof (soundness of if-split). We show that the justification condition for equivalence holds for the if-split rule.

Let $\mathcal{I}$ be a model for the theory TH and t be any closed term. We suppose that t suits the generated tableau $\mathcal{T}_g$ under $\mathcal{I}$, and show that then t suits the required tableau $\mathcal{T}_r$ under $\mathcal{I}$.

If t suits the generated tableau, it must suit at least one of the two rows

$\mathcal{A}$		s
	$\mathcal{G}$	s

We suppose it suits the assertion. Then, for some suiting substitution λ, we have, by the truth condition,

$\mathcal{A}\lambda$ is closed and false under $\mathcal{I}$

and, by the output condition,

$s\lambda$ is closed and equal to t under $\mathcal{I}$.

Let λ_a be a closing substitution for $\mathcal{G}\lambda$. We show that t suits the required tableau $\mathcal{T}_r$ under $\mathcal{I}$, with suiting substitution $\lambda\lambda_a$.

Because $\mathcal{A}\lambda$ is closed, $\mathcal{A}\lambda\lambda_a$ is identical to $\mathcal{A}\lambda$, and hence is closed and false under $\mathcal{I}$. Therefore, by the semantic rule for if-then, (*if* $\mathcal{A}\lambda\lambda_a$ *then* $\mathcal{G}\lambda\lambda_a$) is closed and true under $\mathcal{I}$; that is, the truth condition

(*if* $\mathcal{A}$ *then* $\mathcal{G}$)$\lambda\lambda_a$ is closed and true under $\mathcal{I}$

holds.

Because $s\lambda$ is closed, $s\lambda\lambda_a$ is identical to $s\lambda$, and hence we have the output condition

$s\lambda\lambda_a$ is closed and equal to t under $\mathcal{I}$.

This establishes that t suits the required tableau $\mathcal{T}_r$

	if $\mathcal{A}$ *then* $\mathcal{G}$	s

under $\mathcal{I}$, as we wanted to show.

The proof for the case in which t suits the generated goal is the same. ∎

Simplification

Before we introduce the rules of our system, we would like to describe the simplification process. This is a process in which subexpressions of the tableau are replaced by simpler expressions. Simplification can be applied to the assertions, goals, or output entries of the tableau. Subsentences are replaced by equivalent sentences and subterms are replaced by equal terms. The set of simplifications to be applied depends on the background theory, though some simplifications can be applied in any theory. Because the result of a simplification is always simpler than the given expression, termination of the simplification process is guaranteed.

Simplification is not regarded as a deduction rule. While a rule adds new rows to a tableau without changing those that are already present, simplification replaces an old row with a new one. Also, while the application of a deduction rule is at the discretion of a user or control strategy, the simplification process is mandatory and automatic. That is, we shall fully simplify all the rows of our tableau before applying any deduction rule.

Example (simplification). The and-two simplification

$$\mathcal{F} \wedge \mathcal{F} \Rightarrow \mathcal{F}$$

allows any subsentence of form $(\mathcal{F} \wedge \mathcal{F})$ to be replaced by the corresponding sentence of form $\mathcal{F}$. Applying that simplification, we replace the row

$p(x) \vee \big(q(a) \wedge q(a)\big)$		$g(x)$

with the corresponding row

$p(x) \vee q(a)$		$g(x)$

∎

We arbitrarily divide our simplifications into categories. The true-false simplifications replace subsentences containing instances of the truth symbols *true* or *false*. For example, the and-true simplification

$$\mathcal{F} \wedge \mathit{true} \Rightarrow \mathcal{F}$$

and the not-false simplification

$$\neg\, \mathit{false} \Rightarrow \mathit{true}$$

are true-false simplifications. We provide a full set of these, so that, after simplification, a sentence will contain no proper suboccurrences of the truth symbols *true* or *false*.

There are other *logical* simplifications that are not true-false simplifications, such as the or-two simplification,

$\mathcal{F} \vee \mathcal{F} \Rightarrow \mathcal{F},$

the cond-term-two simplification,

$\textit{if } \mathcal{F} \textit{ then } s \textit{ else } s \Rightarrow s,$

and the all-redundant-quantifier simplification,

$(\forall x)\mathcal{F} \Rightarrow \mathcal{F}, \quad$ where x does not occur free in $\mathcal{F}$.

Finally, there are *theory* simplifications, whose application is limited to a particular theory. For example, if our background theory is the nonnegative integers, we include the plus-zero-right simplification for addition,

$u + 0 \Rightarrow u\,.$

In the theory of strings, we have the left-empty simplification for concatenation,

$\Lambda * v \Rightarrow v\,.$

THE DEDUCTION RULES

We are now ready to introduce the deduction rules of our system. We divide them into several categories.

- *Splitting rules*. Break down a row into its logical components.
- *Resolution rule*. Performs a case analysis on the truth of a subsentence of two rows.
- *Equivalence rule*. Replaces a subsentence with an equivalent sentence.
- *Skolemization rules*. Remove quantifiers.
- *Equality rule*. Replaces a subterm with an equal term.
- *Mathematical induction rule*. Assumes that the desired program behaves correctly on inputs smaller than the given one.

We describe the splitting rules, the resolution rule, the equality rule, and the mathematical induction rule subsequently.

The Splitting Rules

These rules are logically redundant: Any theorem that can be proved with the help of the splitting rules can also be proved without them. Nevertheless, splitting rules often clarify the presentation of a proof.

We include three splitting rules in our system.

Rule (and-split). In tableau notation:

$\mathcal{A}_1 \wedge \mathcal{A}_2$		s
$\mathcal{A}_1$		s
$\mathcal{A}_2$		s

In other words, an assertion that is a conjunction can be decomposed into its two conjuncts. The output entries of the required assertion, if any, are inherited by the two generated assertions. If the required row has no output entry, neither do the generated rows. ∎

The or-split rule is similar.

Rule (or-split).

	$\mathcal{G}_1 \ \vee \ \mathcal{G}_2$	s
	$\mathcal{G}_1$	s
	$\mathcal{G}_2$	s

In other words, a goal that is a disjunction can be decomposed into its two disjuncts. ∎

The and-split and or-split rules reflect the meaning of the tableau: There is an implicit conjunction between the assertions of our tableau and an implicit disjunction between the goals. Note that there is no or-split rule for assertions and no and-split rules for goals.

We have seen the if-split rule.

Rule (if-split).

	if $\mathcal{A}$ *then* $\mathcal{G}$	s
	$\mathcal{G}$	s
$\mathcal{A}$		s

In other words, an implication can be split into an assertion and a goal, its antecedent and consequent, respectively. ∎

The if-split rule reflects the intuitive proof method that, to prove a sentence (*if* $\mathcal{A}$ *then* $\mathcal{G}$), assume the antecedent $\mathcal{A}$ and attempt to prove the consequent $\mathcal{G}$. The justification for the if-split rule was used to illustrate the justification property for equivalence. The justification for the other splitting rules is similar.

Example (if-split rule). Suppose our tableau contains the goal

	if $\epsilon > 0$ *then* $z^2 \leq r \ \wedge \ r < (z+\epsilon)^2$	z

Then we may add its antecedent as an assertion

$\epsilon > 0$		z

and its consequent as the goal

	$z^2 \leq r \ \wedge \ r < (z+\epsilon)^2$	z

∎

The Resolution Rule

The resolution rule is a nonclausal version of the classical Robinson [Ro65] resolution principle introduced for program synthesis (Manna and Waldinger [MW80]); a similar nonclausal resolution rule was developed independently by Murray [Mu82]. The rule corresponds to a case analysis in an informal argument, and it accounts for the introduction of conditional terms in program derivation. We present it first as it applies to two goals.

Rule (GG-resolution)

	$\mathcal{G}_1[\mathcal{P}]$	s
	$\mathcal{G}_2[\mathcal{P}']$	t
	$\mathcal{G}_1\theta[\mathit{false}]$ $\wedge$ $\mathcal{G}_2\theta[\mathit{true}]$	*if* $\mathcal{P}\theta$ *then* $t\theta$ *else* $s\theta$

More precisely, the rule allows the following inference:

- We take $\mathcal{G}_1$ and $\mathcal{G}_2$ to be goal rows with no free variables in common; we rename the variables of these rows to achieve this if necessary.
- We require that $\mathcal{P}$ and $\mathcal{P}'$ be free, quantifier-free subsentences of $\mathcal{G}_1[\mathcal{P}]$ and $\mathcal{G}_2[\mathcal{P}]$, respectively, that are unifiable. We let θ be a most-general unifier of these sentences; thus $\mathcal{P}\theta$ and $\mathcal{P}'\theta$ are identical. In general, there can be more than one subsentence $\mathcal{P}$ in $\mathcal{G}_1[\mathcal{P}]$ and more than one subsentence $\mathcal{P}'$ in $\mathcal{G}_2[\mathcal{P}]$; we take θ to be a most-general unifier of all these subsentences.
- We replace all occurrences of $\mathcal{P}\theta$ in $\mathcal{G}_1\theta$ with *false*, obtaining $\mathcal{G}_1\theta[\mathit{false}]$; we replace all occurrences of $\mathcal{P}'\theta$ (that is, $\mathcal{P}\theta$) in $\mathcal{G}_2\theta$ with *true*, obtaining $\mathcal{G}_2\theta[\mathit{true}]$.
- We take the conjunction of the results, obtaining $(\mathcal{G}_1\theta[\mathit{false}] \wedge \mathcal{G}_2\theta[\mathit{true}])$. After simplification, this is added to the tableau as a new goal.
- The output entry associated with the new goal is the conditional term (*if* $\mathcal{P}\theta$ *then* $t\theta$ *else* $s\theta$). The test of this conditional is the unified subsentence $\mathcal{P}\theta$. The *then*-term and the *else*-term are the appropriate instances $t\theta$ and $s\theta$, respectively, of the output entries of the required goals. ∎

Before discussing the ramifications of this rule, we illustrate it with an example.

Example (resolution rule). We apply the rule to a goal and a copy of itself. Assume our tableau contains the row

	$z^2 \le r \ \wedge \ \neg \boxed{(z+\epsilon)^2 \le r}^{-}$	z

(We shall explain the box and minus-sign annotations subsequently.) This row has the variable z in common with itself; therefore, in the copy we rename z to $\widehat{z}$:

	$\boxed{\widehat{z}^2 \le r}^{+} \wedge \neg\left[(\widehat{z}+\epsilon)^2 \le r\right]$	$\widehat{z}$

The boxed subsentences $\mathcal{P}$: $(z+\epsilon)^2 \le r$ and $\mathcal{P}'$: $\widehat{z}^2 \le r$ are unifiable: A most-general unifier is $\widehat{z} \leftarrow z+\epsilon$. The unified subsentence $\mathcal{P}\theta$ is then $(z+\epsilon)^2 \le r$.

We apply θ to the two rows; the original row is unchanged, but the renamed copy becomes

	$(z+\epsilon)^2 \le r \;\wedge\; \neg\left[\left((z+\epsilon)+\epsilon\right)^2 \le r\right]$	$z+\epsilon$

We replace all copies of $\mathcal{P}\theta$ in the instantiated original row with *false* and all occurrences of $\mathcal{P}\theta$ in the instantiated copy with *true*. The conjunction of the resulting goals is added to the tableau as a new goal

	$z^2 \le r \;\wedge\; (\neg\, \mathit{false})$ $\wedge$ $\mathit{true} \wedge \neg\left[\left((z+\epsilon)+\epsilon\right)^2 \le r\right]$	*if* $(z+\epsilon)^2 \le r$ *then* $z+\epsilon$ *else* z

The output entry of the new goal is a conditional term whose test is the unified subsentence and whose *then*-term and *else*-term are the appropriate instances of the output entries of the two required rows.

The derived row is simplified to

	$z^2 \le r \;\wedge\; \neg\left[(z+2\epsilon)^2 \le r\right]$	*if* $(z+\epsilon)^2 \le r$ *then* $z+\epsilon$ *else* z

The simplifications that were applied to the goal are the true-false simplifications

$\neg\, \mathit{false} \;\rightarrow\; \mathit{true}$

$\mathcal{F} \;\wedge\; \mathit{true} \;\rightarrow\; \mathcal{F}$

$\mathit{true} \;\wedge\; \mathcal{F} \;\rightarrow\; \mathcal{F}$

and the numerical simplification

$(u+v)+v \;\rightarrow\; u+2v\,.$ ∎

Digression (binary search). Let us interrupt the exposition a moment and discuss the intuition behind the step in the preceding example.

The given goal

	$z^2 \le r \;\wedge\; \neg\left[(z+\epsilon)^2 \le r\right]$	z

is a consequence of the initial goal from the derivation of the rational square-root program. It expresses the fact that we would like to find a nonnegative rational number z that is an approximation within ϵ less than the exact square root of r. That is, $\sqrt{r}$ should lie in the half-open interval $[z, z+\epsilon)$. If we succeed, z will be a suitable output for the program.

The derived row

	$z^2 \leq r \ \wedge \ \neg\left[(z+2\epsilon)^2 \leq r\right]$	*if* $(z+\epsilon)^2 \leq r$ *then* $z+\epsilon$ *else* z

expresses the fact that it suffices to find a nonnegative rational z that is a cruder approximation, within 2ϵ less than the exact square root of r. That is, $\sqrt{r}$ should lie in the interval $[z, z+2\epsilon)$. If so, the conditional term

if $(z+\epsilon)^2 \leq r$
then $z+\epsilon$
else z

will be a suitable output for the program.

Why is this? Note that $z+\epsilon$ is the midpoint of the interval $[z, z+2\epsilon)$. In the case in which $(z+\epsilon)^2 \leq r$, that is, $z+\epsilon \leq \sqrt{r}$, we know that $\sqrt{r}$ lies in the right half of the interval. But then the *then*-term $z+\epsilon$ is within ϵ less than the exact square root of r.

In the alternative case, in which $r < (z+\epsilon)^2$, that is, $\sqrt{r} < z+\epsilon$, we know that $\sqrt{r}$ lies in the left half of the interval $[z, z+2\epsilon)$. But then the *else*-term z is already within ϵ less than the exact square root of r.

In either case, the value of the conditional term is within ϵ less than the exact square root, and hence is a suitable output for our program.

The derived row contains the basis for the idea of binary search, while the given row does not. This discovery was obtained by a mechanical step, a single application of the resolution rule. ∎

No-Conditional Cases

In applying the resolution rule, we normally introduce a conditional term as the output entry for the derived row. There are some cases, however, in which we apply the rule without introducing a conditional.

Suppose that the output entries s and t of the required rows happen to be unified by the substitution θ; that is, $s\theta$ and $t\theta$ are identical. In this case, the conditional output entry

if $\mathcal{P}\theta$
then $t\theta$
else $s\theta$

is simplified by the *cond-term-two* simplification

if $\mathcal{F}$ *then* u *else* $u \ \rightarrow \ u$

to yield $s\theta$. Thus, in this case the rule introduces no conditional term at all. The resulting program is of course more efficient than if the conditional had been introduced. Moreover, if the test $\mathcal{P}\theta$ is not primitive, we may not know how to compute the conditional at all.

Suppose now that one of the two required goals, say $\mathcal{G}_2$, has no output entry. Then, instead of the conditional, the output entry for the generated goal will be simply $s\theta$, where s is the output entry for $\mathcal{G}_1$.

Why is this? By the no-output property, the goal $\mathcal{G}_2$ with no output entry is equivalent to one with the new variable u as output entry, where u does not occur free in the row and is unaffected by θ. The output entry generated by the standard, conditional case of the rule is $(\mathit{if}\ \mathcal{P}\theta\ \mathit{then}\ u\theta\ \mathit{else}\ s\theta)$. Because u is unaffected by θ, this is $(\mathit{if}\ \mathcal{P}\theta\ \mathit{then}\ u\ \mathit{else}\ s\theta)$. By the instance property, we may add to our tableau the instance of that row whose goal is the same but whose output entry is $(\mathit{if}\ \mathcal{P}\theta\ \mathit{then}\ s\theta\ \mathit{else}\ s\theta)$, which again simplifies to $s\theta$. We shall call this the one-output case.

Suppose finally that both goals have no output entry; then the derived goal has no output entry either. Why? By the no-output property, again, the first goal is equivalent to one with output entry v, where v is a new variable. By the one-output case of the resolution rule, we may associate with the goal the output entry $v\theta$, that is, v. But then, by the no-output property again, that output entry can be dropped altogether.

The no-conditional cases of the resolution rule will be illustrated after we have introduced the dual versions.

Dual Versions of the Resolution Rule

We have presented the resolution rule as it applies to two goals. With the duality property, we can justify dual versions of the rule, that apply to two assertions, or to an assertion and a goal. These may be expressed as follows.

Rule (AA-resolution).

$\mathcal{A}_1[\mathcal{P}]$		s
$\mathcal{A}_2[\mathcal{P}']$		t
$\mathcal{A}_1\theta[\mathit{false}]$ $\vee$ $\mathcal{A}_2\theta[\mathit{true}]$		$\mathit{if}\ \mathcal{P}\theta$ $\mathit{then}\ t\theta$ $\mathit{else}\ s\theta$

∎

Rule (AG-resolution).

$\mathcal{A}_1[\mathcal{P}]$		s
	$\mathcal{G}_2[\mathcal{P}']$	t
	$\neg\,(\mathcal{A}_1\theta[\mathit{false}])$ $\wedge$ $\mathcal{G}_2\theta[\mathit{true}]$	$\mathit{if}\ \mathcal{P}\theta$ $\mathit{then}\ t\theta$ $\mathit{else}\ s\theta$

∎

Rule (GA-resolution).

	$\mathcal{G}_1[\mathcal{P}]$	s
$\mathcal{A}_2[\mathcal{P}']$		t
	$\mathcal{G}_1\theta[\mathit{false}]$ $\wedge$ $\neg\,(\mathcal{A}_2\theta[\mathit{true}])$	*if* $\mathcal{P}\theta$ *then* $t\theta$ *else* $s\theta$

∎

The notation restrictions and no-conditional cases for these dual versions of the rule are the same as for the original (GG) version.

The justification for these dual versions lies in first pushing the assertions into the goal column by negating them, then applying the GG version of the rule. For the AA version, the resulting goal is subsequently pushed back to the assertion column, negating it once more. The resulting assertion

$$\neg\begin{pmatrix}\neg\,\mathcal{A}_1\theta[\mathit{false}]\\ \wedge\\ \neg\,\mathcal{A}_2\theta[\mathit{true}]\end{pmatrix}$$

is then simplified, with the simplification

$$\neg\,(\neg\,\mathcal{F}\ \wedge\ \neg\,\mathcal{G})\quad\rightarrow\quad\mathcal{F}\ \vee\ \mathcal{G},$$

to yield

$$\begin{array}{c}\mathcal{A}_1\theta[\mathit{false}]\\ \vee\\ \mathcal{A}_2\theta[\mathit{true}].\end{array}$$

The following application of the resolution rule illustrates both the AG version and the one-output case of the rule.

Example (dual version, no-conditional). Suppose our tableau includes the assertion

$\boxed{u \succeq u}^{-}$		

and the goal

	$\boxed{z \succeq a_1}^{+}\ \wedge\ z \succeq a_2$	z

We would like to apply the AG version of the resolution rule to these rows. The two rows have no variables in common. The boxed subsentences are unifiable; a most-general unifier is θ: $\{u \leftarrow a_1,\ z \leftarrow a_1\}$. The result of applying the AG version of the resolution rule is then

	$\neg\,\mathit{false}$ $\wedge$ $\mathit{true}\ \wedge\ a_1 \succeq a_2$	a_1

which simplifies to

	$a_1 \succeq a_2$	a_1

Because the assertion has no output entry, the derived goal has no conditional; this is a one-output case of the rule.

The step illustrated is part of the derivation of a program to find an upper-bound for two objects a_1 and a_2 in the total reflexive theory TR. The intuitive content of the derived row is that, in the case in which $a_1 \succeq a_2$, the term a_1 will be a suitable output for the program. ∎

Polarity

The resolution rule could be applied with the roles of the two rows reversed. For instance, in the preceding section, we applied the AG version of the resolution rule to an assertion and a goal. We could also have applied the GA version to the same goal and assertion, obtaining

	$\mathit{false} \wedge a_1 \succeq a_2$ $\wedge$ $\neg\ \mathit{true}$	a_1

which simplifies to the trivial goal

	false	a_1

It is typical that, of the two ways of applying the rule, one will not advance the proof. In this section, we introduce a syntactic condition that will allow us to avoid many of these fruitless applications of the resolution rule.

Roughly speaking, a subsentence of a tableau is of negative ($-$) or positive ($+$) *polarity* if it is within the scope of an odd or even number, respectively, of negation ($\neg$) connectives. Thus, in the goal

	$\neg\Big([p(x)]^- \ \wedge\ \neg\big([q(y)]^+\big)\Big)$	

$p(x)$ is of negative polarity, because it is within the scope of a single negation, but $q(y)$ is of positive polarity, because it is within the scope of two negations. We have annotated each of these subsentences with its polarity symbol.

We regard the antecedent $\mathcal{F}$ of an implication $(\mathit{if}\ \mathcal{F}\ \mathit{then}\ \mathcal{G})$ as being within the scope of an additional implicit negation, because $(\mathit{if}\ \mathcal{F}\ \mathit{then}\ \mathcal{G})$ is equivalent to $\big((\neg\mathcal{F})\ \mathit{or}\ \mathcal{G}\big)$. Also, while each goal has positive polarity, we regard each assertion $\mathcal{A}$ as having negative polarity, because we could push it into the goal column by negating it, obtaining $(\neg\mathcal{A})$. We regard both sides $\mathcal{F}$ and $\mathcal{G}$ of an equivalence $(\mathcal{F} \equiv \mathcal{G})$ as having both polarities $(\pm)$ because $(\mathcal{F} \equiv \mathcal{G})$ is equivalent to $(\mathit{if}\ \mathcal{F}\ \mathit{then}\ \mathcal{G})\ \wedge\ (\mathit{if}\ \mathcal{G}\ \mathit{then}\ \mathcal{F})$; the first occurrence of $\mathcal{F}$ is within the scope of an additional implicit negation, but the second is not; similarly for $\mathcal{G}$. The *if*-part $\mathcal{F}$ of a conditional sentence $(\mathit{if}\ \mathcal{F}\ \mathit{then}\ \mathcal{G}\ \mathit{else}\ \mathcal{H})$ or a conditional term $(\mathit{if}\ \mathcal{F}\ \mathit{then}\ s\ \mathit{else}\ t)$ also has both polarities.

Example (polarity). The following sentence is annotated according to its polarities

$\left[\begin{array}{l} \mathit{if}\ \big(\mathit{if}\ [p(a)]^{-}\ \mathit{then}\ [q(a)]^{+}\big)^{+} \\ \mathit{then}\ \big([p(a)]^{\pm} \equiv [q(a)]^{\pm}\big)^{-} \end{array}\right]^{-}$		

Because the sentence is an assertion rather than a goal, its polarity, and that of all its subsentences, are reversed. ∎

Now that we have defined polarity of a subsentence of a tableau, we can use the notion to describe a strategy for restricting the resolution rule.

Definition (polarity strategy, for resolution). An application of the resolution rule is in accordance with the *polarity strategy* if at least one negative occurrence of the unified subsentences $\mathcal{P}$ is replaced by *false* and at least one positive occurrence of the unified subsentences $\mathcal{P}'$ is replaced by *true*. The positive and negative occurrences to which we refer may actually have both polarities. ∎

We illustrate the polarity strategy with an example.

Example (polarity strategy). Suppose our tableau contains the two goals

	$\left[\boxed{p(x)}^{\pm} \equiv q(x)\right] \wedge \left(\boxed{p(y)}^{+} \vee \neg\, q(a)\right)$	$t(x, y)$
	$\boxed{p(a)}^{+}$	a

The boxed subsentences are unifiable, with most-general unifier $\{x \leftarrow a, y \leftarrow a\}$. Therefore, we may apply the resolution rule, obtaining

	$\begin{array}{l} [\mathit{false}\ \equiv q(a)] \wedge [\mathit{false}\ \vee \neg\, q(a)] \\ \quad\ \wedge \\ \mathit{true} \end{array}$	$\begin{array}{l} \mathit{if}\ p(a) \\ \mathit{then}\ a \\ \mathit{else}\ t(a, a) \end{array}$

which simplifies to

	$\neg\, q(a)$	$\begin{array}{l} \mathit{if}\ p(a) \\ \mathit{then}\ a \\ \mathit{else}\ t(a, a) \end{array}$

This application of the rule is in accordance with the polarity strategy: The subsentence $p(x)$, which has negative (in fact, both) polarities, is replaced by *false*; also the subsentence $p(a)$, which has positive polarity, is replaced by *true*.

We can also reverse the roles of the two goals in applying the resolution rule, obtaining

	$\begin{array}{l} \mathit{false} \\ \quad\ \wedge \\ [\mathit{true}\ \equiv\ q(a)] \wedge \big(\mathit{true}\ \vee \neg\, q(a)\big) \end{array}$	$\begin{array}{l} \mathit{if}\ p(a) \\ \mathit{then}\ t(a, a) \\ \mathit{else}\ a \end{array}$

which simplfies to the trivial goal

	false	*if* $p(a)$ *then* $t(a,a)$ *else* a

This application of the rule is in violation of the polarity strategy, because no negative occurrence of the unified subsentences is replaced by *false*. ■

We have illustrated the polarity strategy with the GG version of the resolution rule. The strategy is precisely the same for the other versions. We must remember, however, that polarities are reversed in assertions.

Violating the polarity strategy does not always cause us to derive a trivial row; furthermore, observing the strategy does not always prevent us from deriving a trivial row. Nevertheless, it can be shown that observing the polarity strategy never prevents us from completing a proof, and in fact never even lengthens the proof. Because observing the strategy greatly reduces the number of applications of the rule we must consider, there is little reason to ever apply the resolution rule in violation of the polarity strategy.

Relation with Classical Resolution

The question arises as to how the nonclausal resolution rule presented here relates to the classical clausal resolution principle introduced by Robinson [Ro65]. The clausal version of the rule is only applied to assertions that are in clausal form: That is, they are disjunctions of literals, where each literal is either an atom or the negation of an atom. If we apply the clausal resolution principle to the two clauses

$$\mathcal{P} \lor \mathcal{Q}$$
$$\neg\mathcal{P}' \lor \mathcal{R}$$

(where $\mathcal{P}$ is an atom and $\mathcal{Q}$ and $\mathcal{R}$ are themselves clauses) we obtain

$$\mathcal{Q}\theta \lor \mathcal{R}\theta$$

where θ is a most-general unifier of $\mathcal{P}$ and $\mathcal{P}'$.

On the other hand, if we apply the AA version of the resolution rule to the corresponding two assertions

$\mathcal{P} \lor \mathcal{Q}$		
$\neg\mathcal{P}' \lor \mathcal{R}$		

we obtain the new assertion

$\mathit{false} \lor \mathcal{Q}\theta$ $\lor$ $\neg\, \mathit{true} \lor \mathcal{R}\theta$		

which simplifies to

$\mathcal{Q}\theta \lor \mathcal{R}\theta$		

This assertion corresponds to the same clause produced by the classical resolution rule.

Justification of the Resolution Rule

Let us now justify the resolution rule.

Property (soundness of resolution). The resolution rule preserves equivalence of tableaux, and hence is sound, in any theory TH. ▮

Proof (soundness of resolution). Let us reproduce the resolution rule here for convenience:

$$
\begin{array}{c}
\mathcal{T}_r \left\{ \begin{array}{|c|c||c|}
\hline
 & \mathcal{G}_1[\mathcal{P}] & s \\
\hline
 & \mathcal{G}_2[\mathcal{P}'] & t \\
\hline
\end{array} \right. \\
\mathcal{T}_g \left\{ \begin{array}{|c|c||c|}
\hline
 & \begin{array}{c} \mathcal{G}_1\theta[\mathit{false}] \\ \wedge \\ \mathcal{G}_2\theta[\mathit{true}] \end{array} & \begin{array}{l} \mathit{if}\ \mathcal{P}\theta \\ \mathit{then}\ t\theta \\ \mathit{else}\ s\theta \end{array} \\
\hline
\end{array} \right.
\end{array}
$$

We show that the rule satisfies the justification condition for equivalence. Let $\mathcal{I}$ be a model for TH and r be any closed term. We suppose that r suits the generated tableau $\mathcal{T}_g$ under $\mathcal{I}$ and show that r then suits the required tableau $\mathcal{T}_r$ under $\mathcal{I}$.

If r suits $\mathcal{T}_g$ under $\mathcal{I}$, there must be a suiting substitution λ. In other words, by the truth condition,

$$
\begin{pmatrix} \mathcal{G}_1\theta[\mathit{false}] \\ \wedge \\ \mathcal{G}_2\theta[\mathit{true}] \end{pmatrix} \lambda\,, \quad \text{that is,} \quad \begin{pmatrix} \mathcal{G}_1\theta[\mathit{false}]\,\lambda \\ \wedge \\ \mathcal{G}_2\theta[\mathit{true}]\,\lambda \end{pmatrix}
$$

is closed and true under $\mathcal{I}$, and, by the output condition,

$$
\begin{pmatrix} \mathit{if}\ \mathcal{P}\theta \\ \mathit{then}\ t\theta \\ \mathit{else}\ s\theta \end{pmatrix} \lambda\,, \quad \text{that is,} \quad \begin{pmatrix} \mathit{if}\ \mathcal{P}\theta\lambda \\ \mathit{then}\ t\theta\lambda \\ \mathit{else}\ s\theta\lambda \end{pmatrix}
$$

is closed and equal to r under $\mathcal{I}$.

It follows that

$\mathcal{G}_1\theta[\mathit{false}]\,\lambda$ is closed and true under $\mathcal{I}$,

$\mathcal{G}_2\theta[\mathit{true}]\,\lambda$ is closed and true under $\mathcal{I}$,

and $\mathcal{P}\theta\lambda$, $t\theta\lambda$, and $s\theta\lambda$ are all closed.

The proof distinguishes between two cases.

Case: $\mathcal{P}\theta\lambda$ is false under $\mathcal{I}$.

In this case, we show that r suits the first row

$$
\begin{array}{|c|c||c|}
\hline
 & \mathcal{G}_1[\mathcal{P}] & s \\
\hline
\end{array}
$$

of $\mathcal{T}_r$ with suiting substitution $\theta\lambda$.

We must show the truth condition, that

$\mathcal{G}_1[\mathcal{P}]\theta\lambda$ is closed and true under $\mathcal{I}$.

But $\mathcal{G}_1\theta[false]\lambda$ may be obtained from $\mathcal{G}_1[\mathcal{P}]\theta\lambda$ by replacing some occurrences of the closed subsentence $\mathcal{P}\theta\lambda$ with the sentence *false*, which has the same truth-value in this case. Also $\mathcal{G}_1\theta[false]\lambda$ is itself closed and true under $\mathcal{I}$. This implies the desired *truth* condition.

We must also show the output condition, that

$s\theta\lambda$ is closed and equal to r under $\mathcal{I}$.

But the conditional term ($if\ \mathcal{P}\theta\lambda\ then\ t\theta\lambda\ else\ s\theta\lambda$) is, in this case, equal to $s\theta\lambda$ under $\mathcal{I}$. Also the conditional term is closed and equal to r under $\mathcal{I}$. This implies the desired output condition.

Hence, in this case, r suits the first row of $\mathcal{T}_r$ under $\mathcal{I}$. In the alternative case, in which $\mathcal{P}\theta\lambda$ is true under $\mathcal{I}$, we show that r suits the second row of $\mathcal{T}_r$ under $\mathcal{I}$, again with suiting substitution $\theta\lambda$. Hence, in either case, r suits the required tableau $\mathcal{T}_r$ under $\mathcal{I}$. This shows that the rule satisfies the justification condition for equivalence. ∎

The Equality Rule

Normally we describe the properties of the functions and relations of our theory by introducing assertions into the tableau. For example, we may describe the $\succeq$ relation of the total reflexive theory TR by introducing axioms into our tableau as assertions:

$u \succeq u$		
$u \succeq v \ \lor \ v \succeq u$		

Proven properties may also be introduced into the tableau as additional assertions, such as the following property of the upper-bound function ub:

$ub(u, v) \succeq u\ \land$ $ub(u, v) \succeq v$		

This approach is not adequate for describing the equality relation, for which we require a large number of so-called *functional-* and *predicate-substitutivity* axioms, such as

$$\begin{array}{lcl} if\ u = v & \text{and} & if\ u = v \\ then\ f(u, w) = f(v, w) & & then\ p(w, u, x) \equiv p(w, v, x) \end{array}$$

Several such axioms may be required for each function and predicate symbol used in our proof. If we add all the required instances, the strategic ramifications are disastrous: These axioms spawn numerous consequences irrelevant to the theorem at hand.

Most theorem provers successful at working with the equality relation have used special equality rules, rather than representing equality properties axiomatically. The equality rule we use here is a nonclausal version of the paramodulation rule (Wos and G. Robinson [WR69]).

We present the rule first as it applies to two assertions.

Rule (AA-equality).

$\mathcal{A}_1[\ell = r]$		s
$\mathcal{A}_2\langle \ell' \rangle$		t
$\mathcal{A}_1\theta[\mathit{false}]$ $\vee$ $\mathcal{A}_2\theta\langle r\theta \rangle$		*if* $(\ell = r)\theta$ *then* $t\theta$ *else* $s\theta$

More precisely, the rule allows the following inference:

- We take $\mathcal{A}_1$ and $\mathcal{A}_2$ to be assertion rows with no free variables in common; we rename the variables of these rows to achieve this if necessary.
- We require that $\ell = r$ be a subsentence of $\mathcal{A}_1[\ell = r]$ and ℓ' be a subterm of $\mathcal{A}_2\langle \ell' \rangle$ such that ℓ and ℓ' are unifiable, with most-general unifier θ. Here $\ell = r$ and ℓ' are free and quantifier-free subexpressions. As in the resolution rule, there may be many distinct subsentences $\ell = r$ in $\mathcal{A}_1[\ell = r]$ and many subterms ℓ' in $\mathcal{A}_2\langle \ell' \rangle$; the substitution θ must unify all the appropriate expressions.
- We replace all occurrences of $(\ell = r)\theta$ in $\mathcal{A}_1\theta$ with *false*, obtaining $\mathcal{A}_1\theta[\mathit{false}]$; we replace one or more occurrences of $\ell'\theta$ (that is, $\ell\theta$) in $\mathcal{A}_2\theta$ with $r\theta$, obtaining $\mathcal{A}_2\theta\langle r\theta \rangle$. (Because we replace some but not necessarily all occurrences, we use the angle brackets $\langle\ \rangle$ rather than the square brackets $[\ ]$ to denote replacement.)
- We take the disjunction of the results, obtaining $(\mathcal{A}_1\theta[\mathit{false}] \vee \mathcal{A}_2\theta\langle r\theta \rangle)$. After simplification, this is added to the tableau as a new assertion.
- The output entry associated with the new assertion is the conditional term $\big(\mathit{if}\ (\ell = r)\theta\ \mathit{then}\ t\theta\ \mathit{else}\ s\theta\big)$. ∎

We have presented the equality rule as it applies to two assertions. As with the resolution rule, we can apply dual versions of the equality rule to an assertion and a goal, or to two goals; the justification of these versions of the rule appeals to the duality property.

Also as with the resolution rule, we introduce a conditional term into the output entry only if both given rows have output entries that fail to be unified by the substitution θ. If only one of the rows has an output entry s, we take $s\theta$ as the new output entry. If both rows have output entries s and t that are unified by θ, we take the unified term $s\theta$ as the new output entry. If both rows have no output entry, neither will the new row.

An application of the rule is in accordance with the polarity strategy if at least one negative occurrence of an equality $\ell = r$ is replaced by *false*; no restriction is imposed on the occurrences of the subterms ℓ'.

The equality rule allows us to replace instances of the left term ℓ with corresponding instances of the right term r. By the symmetry of the equality relation, we can justify a right-to-left version of the rule, which allows us to replace instances of the right term r with corresponding instances of the left term ℓ.

We illustrate the equality rule with an example.

Example (equality rule). This example is taken from the transformation of a program to reverse a string. We are in the process of deriving an auxiliary subprogram $rev2(s, t)$ to reverse the string s and concatenate it onto the string t.

Our tableau contains the two goals

	$\neg\big(\boxed{s} = \Lambda\big)^{-}$	$rev2\big(tail(s),\ head(s) \cdot t\big)$
	$z = rev\big(\boxed{s}\big) * t$	z

These rows have no variables in common. The boxed subterms are identical and hence unifiable with most-general unifier $\{\ \}$. The result of applying a dual version of the rule, the GG-equality rule, is then

	$\neg\ false$ $\land$ $z = rev(\Lambda) * t$	$if\ \ s = \Lambda$ $then\ \ z$ $else\ \ rev2\big(tail(s),\ head(s) \cdot t\big)$

which reduces under simplification to

	$z = t$	$if\ \ s = \Lambda$ $then\ \ z$ $else\ \ rev2\big(tail(s),\ head(s) \cdot t\big)$

Because both terms have output entries, a conditional term is introduced as the new output entry. The application is in accordance with the polarity strategy, because the occurrence of the equality $(s = \Lambda)$ is negative in the tableau. ∎

Example (equality rule). This example is taken from the derivation of a square-root program in the theory of nonnegative rationals. We assume our tableau contains the assertion

$\big(\boxed{0 \cdot v} = 0\big)^{-}$		

which is an axiom for multiplication, and the goal

	$\boxed{z \cdot z} \le r\ \ \land$ $r < (z + \epsilon)^2$	z

The two rows have no variables in common. The boxed subterms are unifiable; a most-general unifier is $\{z \leftarrow 0,\ v \leftarrow 0\}$. The result of applying a dual version of the equality rule is then

	$\neg\ false$ $\land$ $0 \le r\ \land$ $r < (0 + \epsilon)^2$	0

which reduces under simplification to

	$r < \epsilon^2$	0

(The condition $0 \leq r$ is simplified to *true* in the theory of nonnegative rationals.) Because the given assertion has no output entry, no conditional construct is introduced in applying the rule. The application is in accordance with the polarity strategy, because the occurrence of the equality $(0 \cdot v = 0)$ is negative in the tableau.

The intuitive content of the derived goal is that, for the case in which $r < \epsilon^2$, that is, in which $\sqrt{r}$ is in the half-open interval $[0, \epsilon)$, we know 0 is a suitable output for the desired square-root program. ∎

The equality rule allows us to discard all the equality axioms, except for the reflexivity axiom $u = u$, from our initial tableau, without sacrificing the possibility of completing any derivation.

The Well-Founded Induction Rule

The well-founded induction principle is valuable for program synthesis and other applications because of its generality: The induction principles of all theories turn out to be instances of the well-founded induction rule. In derivation proofs, use of the rule corresponds to the introduction of recursion, or other repetitive constructs, into the derived program. Before we describe the rule, we introduce the notion of a well-founded relation.

Definition (well-founded relation). A relation $\prec$ is *well-founded* (in a theory TH) if there are no infinite decreasing sequences in TH, i.e., no sequences $x_1, x_2, x_3, \ldots$ such that

$$x_1 \succ x_2 \quad \text{and} \quad x_2 \succ x_3 \quad \text{and} \quad \ldots$$ ∎

For example, the less-than relation $<$ and the proper-substring relation $\prec_{string}$ are well-founded in the theories of nonnegative integers and strings, respectively. (A string s is a proper substring of a string t, written $s \prec_{string} t$, if s and t are distinct and if the elements of s occur contiguously in t.) On the other hand, the less-than relation $<$ is not well-founded in the theory of nonnegative rationals, because $1, 1/2, 1/4, 1/8, \ldots$ constitutes an infinite decreasing sequence under $<$.

Well-founded relations are of interest to us because of the following property.

Property (well-founded induction principle). For any well-founded relation $\prec$ in theory TH and any sentence $\mathcal{P}[x]$, any closed instance of the following sentence is valid in TH:

$$\begin{array}{l} \textit{if } (\forall x') \begin{bmatrix} \textit{if } (\forall x) \begin{bmatrix} \textit{if } x \prec x' \\ \textit{then } \mathcal{P}[x] \end{bmatrix} \\ \textit{then } \mathcal{P}[x'] \end{bmatrix} \\ \textit{then } (\forall x)\mathcal{P}[x] \end{array}$$

where x' does not occur free in $\mathcal{P}[x]$. ∎

In other words, suppose we are trying to prove that $\mathcal{P}[x]$ is true for every object x. For this purpose, it suffices to consider an arbitrary object x' and show that $\mathcal{P}[x']$ holds under the induction hypothesis that $\mathcal{P}[x]$ is true for every x such that $x \prec x'$. The well-founded induction principle is called *complete* induction or *course-of-values* induction when $\prec$ is taken to be the less-than relation $<$ over the nonnegative integers. It is also called *noetherian* induction.

In the deductive-tableau framework, this principle is represented as a rule.

Rule (well-founded induction).

assertions	goals	$f(a)$
	$\mathcal{Q}[a, z]$	z
if $x \prec_w a$ *then* $\mathcal{Q}[x, f(x)]$		

Here $\mathcal{Q}[a, z]$ is the initial goal of the tableau; we require that z be the only free variable in the row. (If there are several output entries $z_1, \ldots, z_n$, all of them may occur free in the goal.) The relation $\prec_w$ is required to be well-founded in TH. The function symbol f stands for the function we are trying to compute. ∎

The rationale for the induction rule is as follows. We are trying to construct a program to compute a function f that, for a given input a, will yield an output z that satisfies the input-output relation $\mathcal{Q}[a, z]$. It suffices to conduct the derivation under the induction hypothesis that the function f will behave properly on each input x that is less than a under $\prec_w$. More precisely, we may assume inductively that, for each input x such that $x \prec_w a$, the output $f(x)$ will satisfy the input-output relation $\mathcal{Q}[x, f(x)]$.

Example (well-founded induction rule). Recall that the initial goal for the *front-last* derivation is

assertions	goals	*front*(s)	*last*(s)
	if $\neg(s = \Lambda)$ *then* $char(z_2) \wedge$ $s = z_1 * z_2$	z_1	z_2

This row says that we would like our program to decompose a nonempty string s into the concatenation of a string z_1 and a character z_2, so that *front*(s) and *last*(s) can be taken to be z_1 and z_2, respectively.

According to the induction rule, we may add to our tableau the new assertion

if $x \prec_w s$ *then if* $\neg(x = \Lambda)$ *then* $char(last(x)) \wedge$ $x = front(x) * last(x)$			

This row corresponds to the induction hypothesis that, for any nonempty input x less than s under $\prec_w$, the functions *front* and *last* will indeed decompose x into the concatenation of string $front(x)$ and character $last(x)$. The relation $\prec_w$ can be any well-founded relation. ∎

When the program being derived has more than one input, the well-founded relation $\prec_w$ applies to two tuples of inputs, rather than to the inputs themselves.

Example (well-founded induction rule). The initial goal for the rational square-root derivation is

assertions	goals	$sqrt(r, \epsilon)$
	$if\ \epsilon > 0$ $then\ z^2 \leq r \wedge$ $r < (z + \epsilon)^2$	z

According to the well-founded induction rule, we may add to our tableau as an induction hypothesis the assertion

$if\ \langle x, v\rangle \prec_w \langle r, \epsilon\rangle$ $then\ if\ v > 0$ $then\ \big(sqrt(x, v)\big)^2 \leq x \wedge$ $x < \big(sqrt(x, v) + v\big)^2$		

This now declares that the square-root program behaves properly for any pairs of inputs less than the original inputs under $\prec_w$. The well-founded relation applies to two pairs, that is, two 2-tuples, rather than to two individual nonnegative rationals. ∎

Recursion Formation

The induction hypothesis introduced by application of the induction rule contains occurrences of the function symbol f, which denotes the function we are trying to compute. If the induction hypothesis is used in the proof, it can happen that terms of form $f(t)$ will be introduced into the output column and hence into the derived program. This is the mechanism by which recursive calls are introduced into the program.

Example (recursion formation). We have applied the induction rule to the initial goal of the *front-last* derivation, introducing the induction hypothesis

$if\ x \prec_w s$ $then\ if\ \neg(x = \Lambda)$ $then\ char\big(last(x)\big) \wedge$ $\Big(x = \boxed{front(x) * last(x)}\Big)^-$			

This induction hypothesis contains occurrences of the function symbols *front* and *last*, which we are trying to compute. Suppose we have also derived the following goal:

	$char(u) \wedge$ $char(z_2) \wedge$ $s = u \cdot \boxed{\widehat{z}_1 * z_2}$	$u \cdot \widehat{z}_1$	z_2

Note that the boxed subterms of the two rows are unifiable, with most-general unifier $\{\widehat{z}_1 \leftarrow front(x),\ z_2 \leftarrow last(x)\}$. By application of the right-to-left version of the equality rule, we obtain, after simplfication, the goal

	$x \prec_w s \wedge$ $\neg\,(x = \Lambda) \wedge$ $char(u) \wedge$ $char\big(last(x)\big) \wedge$ $s = u \cdot x$	$u \cdot front(x)$	$last(x)$

By using the induction hypothesis, we have introduced the terms *front*(x) and *last*(x) into the output column. This will result in the formation of recursive calls in the final program.

The condition $x \prec_w s$ in the goal has the effect of ensuring that these recursive calls will not cause a nonterminating computation of the final program. If there were an infinite sequence of calls to either *front* or *last*, the corresponding arguments would constitute an infinite sequence of strings decreasing with respect to $\prec_w$; this would contradict the well-foundedness of $\prec_w$.

The condition $\neg\,(x = \Lambda)$ in the goal guarantees that the argument to the recursive calls is a legal input, i.e., that it is nonempty.

The relation $\prec_w$ to be used in the proof has not been determined; it may be any well-founded relation. ∎

We illustrate recursion formation with another example.

Example (recursion formation). In the derivation of the rational square-root program, suppose we have derived the assertion

if $\langle x, v\rangle \prec_w \langle r, \epsilon\rangle$ *then if* $v > 0$ *then* $\boxed{\big(sqrt(x,v)\big)^2 \le x \wedge \neg\big[\big(sqrt(x,v)+v\big)^2 \le x\big]}^{-}$			

This is an immediate consequence of our induction hypothesis.

We have earlier obtained the goal

	$\boxed{z^2 \le r \wedge \neg\big[(z+2\epsilon)^2 \le r\big]}^{+}$		*if* $(z+\epsilon)^2 \le r$ *then* $z+\epsilon$ *else* z

This was obtained by an application of the resolution rule in a previous example. The boxed subsentences of the two rows are unifiable, with most-general unifier $\{x \leftarrow r,\ v \leftarrow 2\epsilon,\ z \leftarrow sqrt(r, 2\epsilon)\}$. By application of the resolution rule, we obtain, after simplification, the goal

	$\langle r, 2\epsilon\rangle \prec_w \langle r, \epsilon\rangle \wedge$ $2\epsilon > 0$	*if* $\left(sqrt(r,2\epsilon)+\epsilon\right)^2 \le r$ *then* $sqrt(r,2\epsilon)+\epsilon$ *else* $sqrt(r,2\epsilon)$

By using the induction hypothesis in the proof, we have introduced three occurrences of the recursive call $sqrt(r, 2\epsilon)$ into the output column. The condition $\langle r, 2\epsilon\rangle \prec_w \langle r, \epsilon\rangle$ in the goal guarantees that these recursive calls do not lead to a nonterminating computation. The condition $2\epsilon > 0$ guarantees that the arguments r and 2ϵ of the recursive calls are legal inputs, that is, 2ϵ is positive. The well-founded relation $\prec_w$ is yet to be determined. ∎

Choice of a Well-Founded Relation

There are many well-founded relations that can serve as the basis for an induction proof. Until the proof is well under way, it may be difficult to determine which relation will be most convenient to use. Rather than attempting to choose a relation at the beginning of the proof, we prefer to start the proof with an unspecified relation $\prec_w$, so that we can discover those properties the relation is required to satisfy.

We assume that a number of relations are given in advance to be well-founded, with certain known properties. In addition, there are mechanisms for constructing new well-founded relations from old ones, to satisfy certain properties. When the required properties of the unspecified relation $\prec_w$ match the properties of a known or constructed relation $\prec_r$, we can choose $\prec_w$ to be that relation $\prec_r$.

Example (choice of a well-founded relation). In the theory of strings, the proper substring relation $\prec_{string}$ is given to be well-founded and known to have the property that the tail of a nonempty string is its proper substring, that is,

if $\neg(y = \Lambda)$ *then* $\boxed{tail(y) \prec_{string} y}^{-}$		

In a derivation of *front-last*, we obtain the goal

	$\boxed{tail(s) \prec_w s}^{+} \wedge$ $\neg\left(tail(s) = \Lambda\right)$	$head(s)\cdot$ $front\left(tail(s)\right)$	$last\left(tail(s)\right)$

This suggests that we take the relation $\prec_w$ to be the proper substring relation $\prec_{string}$. We can then apply the resolution rule to these two rows, with most-general unifier $\{y \leftarrow s\}$, to obtain, after simplification, the goal:

	$\neg(s = \Lambda) \wedge$ $\neg(tail(s) = \Lambda)$	$head(s) \cdot$ $front(tail(s))$	$last(tail(s))$

∎

The selection of the well-founded relation may be regarded as an extralogical step, to be performed by an external mechanism. Alternatively, we can extend our theories to include well-founded relations as objects. We may then regard $x \prec_w y$ as an abbreviation for $\prec(w, x, y)$, where w is a variable that ranges over well-founded relations. In the above resolution step, when we unified $tail(y) \prec_{string} y$ with $tail(s) \prec_w s$, the unification algorithm would then include $w \leftarrow string$ as a replacement in the most-general unifier. In other words, the choice of well-founded relation would be a byproduct of the proof process. ∎

EXAMPLES

In this section we give some examples of the derivation of specific programs.

The Front-Last Derivation

We have not given all the rules in the system, but we have shown enough to illustrate a full derivation of the *front-last* program. This program, the reader will recall, is to find, for a given nonempty string s, two outputs: the last character $last(s)$ of s and the string $front(s)$ of all but the last character of s. The specification is

$$\langle front(s), last(s)\rangle \Leftarrow \begin{cases} \textit{find } \langle z_1, z_2\rangle \text{ such that} \\ \quad \textit{if } \neg(s = \Lambda) \\ \quad \textit{then } char(z_2) \ \wedge \ s = z_1 * z_2 \end{cases}$$

in the theory of strings. Our initial goal is thus

assertions	goals	$front(s)$	$last(s)$
	1. $\textit{if } \neg(s = \Lambda)$ $\textit{then } char(z_2) \wedge s = z_1 * z_2$	z_1	z_2

Properties of the theory of strings, expressed as assertions, are present in the initial tableau, and will be mentioned as we use them.

By the if-split rule, we may decompose our goal into its antecedent and consequent

2. $\neg(s = \Lambda)$			
	3. $char(z_2) \wedge$ $s = \boxed{z_1 * z_2}$	z_1	z_2

The output entries z_1 and z_2 have been dropped from the row 2, because these variables do not occur free in the assertion. We have annotated goal 3 in anticipation of a future step.

The Base Case

By the equality rule, applied to an axiom for concatenation,

$\left(\boxed{\Lambda * y} = y\right)^{-}$			

and the most recent goal, with most-general unifier $\{z_1 \leftarrow \Lambda,\ y \leftarrow z_2\}$, we obtain the goal

	4. $char(z_2) \wedge$ $\boxed{s = z_2}^{+}$	Λ	z_2

Note that the first output entry has been instantiated.

By the resolution rule, applied to the reflexivity axiom

$\boxed{x = x}^{-}$			

and the goal, with most-general unifier $\{x \leftarrow s,\ z_2 \leftarrow s\}$, we obtain

	5. $char(s)$	Λ	s

Now both output entries have been instantiated. The intuitive content of this row is that in the case in which the input string s consists of a single character, $front(s)$ may be taken to be Λ and $last(s)$ to be s itself. This will lead to the base case for the program we are constructing. Let us set it aside for a while and turn our attention to the recursive case.

The Recursive Case

We have earlier developed the goal

	3. $char(z_2) \wedge$ $s = \boxed{z_1 * z_2}$	z_1	z_2

By the equality rule, applied to an axiom for concatenation,

if $char(u)$ *then* $\left(\boxed{(u \cdot y_1) * y_2} = u \cdot (y_1 * y_2)\right)^{-}$			

and the goal, with most-general unifier $\{z_1 \leftarrow u \cdot y_1, z_2 \leftarrow y_2\}$, we obtain

	6. $char(u) \wedge$ $char(y_2) \wedge$ $s = u \cdot \boxed{y_1 * y_2}$	$u \cdot y_1$	y_2

By the induction rule, applied as always to the initial goal, we may assume the induction hypothesis

7. $\mathit{if}\ x \prec_w s$ $\mathit{then\ if}\ \neg(x = \Lambda)$ $\mathit{then}\ char\big(last(x)\big) \wedge$ $\Big(x = \boxed{front(x) * last(x)}\Big)^-$			

By the equality rule, applied right-to-left to assertion 7 and goal 6, with most-general unifier $\{y_1 \leftarrow front(x), y_2 \leftarrow last(x)\}$, we obtain the goal

	8. $x \prec_w s \wedge$ $\neg(x = \Lambda) \wedge$ $char(u) \wedge$ $\boxed{char\big(last(x)\big)}^+ \wedge$ $s = u \cdot x$	$u \cdot front(x)$	$last(x)$

Note that, by use of the induction hypothesis, the recursive calls $front(x)$ and $last(x)$ have been introduced into the output columns.

We next apply the resolution rule, again to the induction hypothesis and the goal. Because these rows have the variable x in common, we rename the variable in the induction hypothesis:

7. $\mathit{if}\ x' \prec_w s$ $\mathit{then\ if}\ \neg(x' = \Lambda)$ $\mathit{then}\ \boxed{char\big(last(x')\big)}^- \wedge$ $x' = front(x') * last(x')$			

Applying the rule, with most-general unifier $\{x' \leftarrow x\}$, we obtain

	9. $x \prec_w s \wedge$ $\neg(x = \Lambda) \wedge$ $char(u) \wedge$ $\boxed{s = u \cdot x}^+$	$u \cdot front(x)$	$last(x)$

By the resolution rule, applied to the *decomposition* property for strings,

$\mathit{if}\ \neg(y = \Lambda)$ $\mathit{then}\ \boxed{y = head(y) \cdot tail(y)}^-$			

and the goal, taking most-general unifier $\{y \leftarrow s,\ u \leftarrow head(s),\ x \leftarrow tail(s)\}$, we obtain

	10. $\neg(s = \Lambda) \wedge$ $\boxed{tail(s) \prec_w s}^+ \wedge$ $\neg\big(tail(s) = \Lambda\big) \wedge$ $char\big(head(s)\big)$	$head(s) \cdot front\big(tail(s)\big)$	$last\big(tail(s)\big)$

Note that at this stage the output entries are fully instantiated. It remains to select the well-founded relation and to combine the base case and the recursive calls.

Choice of a Well-Founded Relation

Let us assume we know the following property of the proper substring relation $\prec_{string}$:

if $\neg\,(y = \Lambda)$ *then* $\boxed{tail(y) \prec_{string} y}^{-}$			

By application of the resolution rule, taking $\prec_w$ to be the $\prec_{string}$ and taking the most-general unifier to be $\{y \leftarrow s\}$, we obtain

	11. $\neg\,(s = \Lambda)\ \wedge$ $\neg\,\big(tail(s) = \Lambda\big)\ \wedge$ $\boxed{char\big(head(s)\big)}^{+}$	$head(s) \cdot$ $front\big(tail(s)\big)$	$last\big(tail(s)\big)$

This step has suggested that the well-founded relation be taken to be the proper substring relation.

The Final Steps

By the resolution rule, applied to the axiom

if $\neg(y = \Lambda)$ *then* $\boxed{char\big(head(y)\big)}^{-}$			

and the goal, with most-general unifier $\{y \leftarrow s\}$, we obtain

	12. $\neg\,(s = \Lambda)\ \wedge$ $\boxed{\neg\,(tail(s) = \Lambda)}^{+}$	$head(s) \cdot$ $front\big(tail(s)\big)$	$last\big(tail(s)\big)$

By the resolution rule, applied to the *trichotomy* property of strings,

$y = \Lambda\ \vee\ char(y)\ \vee\ \boxed{\neg\,(tail(y) = \Lambda)}^{-}$			

with most-general unifier $\{y \leftarrow s\}$, we obtain

	13. $\boxed{\neg\,(s = \Lambda)}^{+}\ \wedge$ $\neg\,char(s)$	$head(s) \cdot$ $front\big(tail(s)\big)$	$last\big(tail(s)\big)$

By the resolution rule, applied to our earlier assertion

2. $\boxed{\neg(s = \Lambda)}^{-}$			

and the goal, we obtain

	14. $\neg\ \boxed{char(s)}^{-}$	$head(s) \cdot$ $front(tail(s))$	$last(tail(s))$

Finally, by the resolution rule applied to this goal and the one we set aside for the base case,

	5. $\boxed{char(s)}^{+}$	Λ	s

we obtain the goal

	15. *true*	*if* $char(s)$ *then* Λ *else* $head(s) \cdot front(tail(s))$	*if* $char(s)$ *then* s *else* $last(tail(s))$

By this step, conditional terms have been introduced into our program.

Because we have obtained the goal *true* with primitive output entries, we can take this to be the final goal of our tableau. The program we extract from the proof is

$$front(s) \Leftarrow \begin{cases} \textit{if } char(s) \\ \textit{then } \Lambda \\ \textit{else } head(s) \cdot front(tail(s)) \end{cases}$$

$$last(s) \Leftarrow \begin{cases} \textit{if } char(s) \\ \textit{then } s \\ \textit{else } last(tail(s)) \ . \end{cases}$$

The Final Square-Root Program

We do not give the full derivation for the square-root program we have been using as an example; it is described in Manna and Waldinger [MW87a]. The final program we obtain is

$$sqrt(r, \epsilon) \Leftarrow \begin{cases} \textit{if } max(r, 1) < \epsilon \\ \textit{then } 0 \\ \textit{else if } \left(sqrt(r, 2\epsilon) + \epsilon\right)^2 \le r \\ \qquad \textit{then } sqrt(r, 2\epsilon) + \epsilon \\ \qquad \textit{else } sqrt(r, 2\epsilon) \end{cases}$$

Let us explain this somewhat odd program because it illustrates a more general phenomenon.

Recall that the program is intended to find a rational approximation $sqrt(r, \epsilon)$ that is within ϵ less than the exact square root of r; that is, $\sqrt{r}$ is to belong to the half-open interval $\left[sqrt(r, \epsilon),\ sqrt(r, \epsilon)+\epsilon\right)$.

In the case in which the error tolerance is quite large, that is, $max(r, 1) < \epsilon$, it turns out that $r < \epsilon^2$, that $\sqrt{r}$ belongs to the interval $[0, \epsilon)$ and, hence, that 0 is a good enough approximation to the square root of r.

Otherwise, we double our error tolerance and recursively find an approximation $sqrt(r, 2\epsilon)$ that is within 2ϵ less than the square root of r; that is, $\sqrt{r}$ belongs to the interval $\left[sqrt(r, 2\epsilon),\ sqrt(r, 2\epsilon)+2\epsilon\right)$. The program then asks whether $\left(sqrt(r, 2\epsilon) + \epsilon\right)^2 \leq r$, that is, whether $\sqrt{r}$ is in the right or the left half of our interval.

In the case in which $\sqrt{r}$ is in the right half $\left[sqrt(r, 2\epsilon) + \epsilon,\ sqrt(r, 2\epsilon) + 2\epsilon\right)$, we can take $sqrt(r, 2\epsilon) + \epsilon$ to be our approximation to the square root; it is certain to be within ϵ less than $\sqrt{r}$.

In the alternative case, in which $\sqrt{r}$ is in the left half $\left[sqrt(r, 2\epsilon),\ sqrt(r, 2\epsilon) + \epsilon\right)$, we can take $sqrt(r, 2\epsilon)$ itself to be our approximation. In either case, the conditional expression will yield an approximation within ϵ less than $\sqrt{r}$.

This recursive program uses a binary-search technique but it does not resemble conventional iterative binary-search algorithms. Usually a binary-search algorithm will begin with a very large interval containing the desired output. It will divide the interval in half at each iteration and will retain the half that contains the output. The process continues until the interval is sufficiently small, that is, shorter than a given error tolerance.

Rather than dividing an interval in half at each iteration, our derived program doubles its error tolerance at each recursion, until the tolerance is quite large. At this point, it can form a large interval which contains the desired output. As it unwinds from the recursion, it implicitly divides this interval in half, just as a conventional algorithm does. Similar recursive binary-search programs may be obtained for division and other numerical problems.

This program was first derived by purely formal manipulation of the rules of the system, to explore the search space, without any expectation of finding a program of this form. When the program was obtained, we did not understand it and thought we had made an error in the derivation.

The program as derived is quite inefficient, as it contains several occurrences of the same recursive call $sqrt(r, 2\epsilon)$. These can be replaced by a single recursive call by ordinary elimination of common subexpressions. More sophisticated program transformation techniques (Harrison and Khoshnevisan [HK86]) have been applied to transform the program into a linear iterative form.

The Slowsort Program

Another example of a program obtained by formal manipulation is this sorting program obtained by Traugott [Tr89].

$$sort(l) \Leftarrow \begin{cases} \mathit{if}\ l = \langle\,\rangle \\ \mathit{then}\ \langle\,\rangle \\ \mathit{else\ if}\ tail(l) = \langle\,\rangle \\ \quad \mathit{then}\ l \\ \quad \mathit{else\ if}\ head(l) \le head\big(sort\big(tail(l)\big)\big) \\ \quad\quad \mathit{then}\ head(l) \cdot sort\big(tail(l)\big) \\ \quad\quad \mathit{else}\ head\big(sort\big(tail(l)\big)\big)\ \cdot \\ \quad\quad\quad sort\big(head(l) \cdot tail\big(sort\big(tail(l)\big)\big)\big) \end{cases}$$

Here l is a list of numbers and $\langle\,\rangle$ is the empty list. No particular claims are made for the efficiency of this program. For example, to find the minimum element of $tail(l)$, the program sorts it and throws away all but the first element. The program is unusual in that it sorts the list without invoking any auxiliary programs, just basic list-processing primitives.

Traugott derived other sorting programs with this property as well. He also considered the relationship between the proof strategy and the form of the extracted program.

SUBPROGRAMS

Once we have derived a program f, we can use it as a subprogram in future derivations. We do this by including in the tableau for these derivations an assertion stating that the derived program f does indeed meet its specification.

More precisely, suppose we have derived a program $f(a) \Leftarrow t$ to meet a specification

$$f(a) \ \Leftarrow\ \mathit{find}\ z\ \mathit{such\ that}\ \mathcal{Q}[a, z]\,.$$

Then in the initial tableau for the derivation of a new program g, we may include the assertion

$(\forall a)\,\mathcal{Q}[a, f(a)]$		

which states that f does satisfy its specification. If this assertion is used in the proof, the new program g may invoke the earlier program f. The function symbol f is included in the primitive list for the derivation of g.

If we choose, we may include the program f itself as an assertion in the derivation for g. That is, we may include the assertion

$(\forall a)[f(a) = t]$		

in the initial tableau for g. If we do this we have lost a certain degree of modularity, because the program for g may depend on the particular implementation for f. We thus are no longer free to replace the program for f with a different program meeting the same specification.

Program Transformation

At this point we illustrate both the formation of subprograms and the application of deductive methods to program transformation problems.

We suppose we are given a program

$$reverse(s) \;\Leftarrow\; \begin{cases} \mathit{if}\ \ s = \langle\,\rangle \\ \mathit{then}\ \ s \\ \mathit{else}\ reverse\big(tail(s)\big) * head(s) \end{cases}$$

for reversing the characters of a given string s. The program is inefficient because, in executing successive recursive calls to *reverse*, it will be computing the concatenation function many times.

To transform a given program, we may regard that program as the specification for a new program. For this example, the new specification is

$$reverse1(s) \;\Leftarrow\; \mathit{find}\ z\ \mathit{such\ that}\ z = reverse(s)\,.$$

Of course the *reverse* program itself will satisfy this specification, but different derivations will yield different programs, some of them more efficient than others.

Looking ahead, the particular program we shall derive is

$$reverse1(s) \;\Leftarrow\; reverse2(s, \Lambda)$$

where

$$reverse2(s,t) \;\Leftarrow\; \begin{cases} \mathit{if}\ \ s = \Lambda \\ \mathit{then}\ t \\ \mathit{else}\ reverse2\big(tail(s),\ head(s) \cdot t\big) \end{cases}$$

The auxiliary subprogram $reverse2(s,t)$ may be regarded as a generalization of *reverse*. It meets the specification

$$reverse2(s,t) \;\Leftarrow\; \mathit{find}\ z\ \mathit{such\ that}\ z = reverse(s) * t.$$

In other words, it reverses s and concatenates the result with t. To complete the derivation of *reverse1*, it is necessary to derive *reverse2*.

Derivation of Reverse2

We will not give the full derivation of *reverse2*, but will present those steps relevant to our present discussion.

We begin with the initial tableau

assertions	goals	$reverse2(s,t)$
	$z = reverse(s) * t$	z

The function symbol *reverse* is excluded from the primitive list, so that it may not occur in the derived program. By the well-founded induction rule, we may assume the induction hypothesis

$\mathit{if}\ \langle x,\ y\rangle \prec_w \langle s,\ t\rangle$ $\mathit{then}\ \boxed{reverse2(x,y) = reverse(x) * y}^{-}$		

¿From our initial goal, the definition of *reverse*, and some properties of strings, we eventually obtain the goal

	$\neg(s = \Lambda) \ \wedge$ $\boxed{z = reverse\big(tail(s)\big) * \big(head(s) \cdot t\big)}^{+}$	z

The boxed subsentences of the assertion and the goal unify, with most-general unifier $\{x \leftarrow tail(s),\ y \leftarrow head(s) \cdot t,\ z \leftarrow reverse2\big(tail(s),\ head(s) \cdot t\big)\}$. By the resolution rule, we obtain

	$\langle tail(s),\ head(s) \cdot t\rangle \prec_w \langle s,\ t\rangle \ \wedge$ $\neg(s = \Lambda)$	$reverse2\big(tail(s),\ head(s) \cdot t\big)$

Use of the induction hypothesis has accounted for the introduction of the recursive call $reverse2\big(tail(s), head(s) \cdot t\big)$ into the output entry and, ultimately, into the *reverse2* program.

Derivation of Reverse1

Once we have derived *reverse2*, the derivation of the program *reverse1* is simple. We begin with the initial tableau.

assertions	goals	$reverse1(s)$
	$\boxed{z = reverse(s)}^{+}$	z

We may include in our tableau the assertion that, for all x and y, the program $reverse2(x, y)$ does meet the specification from which it was derived:

$reverse2(x, y) =$ $\boxed{reverse(x) * y}$		

We assume that we have the following property of concatenation:

$\Big(\boxed{\widehat{x} * \Lambda} = \widehat{x}\Big)^{-}$		

By the equality rule applied to this and the previous assertion, with most-general unifier $\{\widehat{x} \leftarrow reverse(x),\ y \leftarrow \Lambda\}$, we obtain

$\boxed{reverse2(x, \Lambda) = reverse(x)}^{-}$		

By the resolution rule, applied to this assertion and the initial goal, with most-general unifier $\{x \leftarrow s,\ z \leftarrow reverse2(s, \Lambda)\}$, we obtain the final goal

	$true$	$reverse2(s, \Lambda)$

¿From this proof, we extract the program

$$reverse1(s) \ \Leftarrow \ reverse2(s, \Lambda)\,.$$

The Need for Generalization

This derivation illustrates a phenomenon in program synthesis that reflects a corresponding observation in theorem proving. It has been remarked that in proving a theorem by induction, it is often necessary to prove a more general theorem, so as to have the benefit of a more general induction hypothesis. (This fact has been exploited by the Boyer-Moore theorem prover [BM79].) Similarly, in deriving a program, it is sometimes necessary to derive a more general program, so as to have the benefit of a more general recursive call.

To illustrate this phenomenon, let us see what would have happened had we begun the derivation of the *reverse1* program without first deriving *reverse2*. We begin with the goal

assertions	goals	$reverse1(s)$
	$z = reverse(s)$	z

By the well-founded induction rule, we may assume the induction hypothesis

$\mathit{if}\ x \prec_w s$ $\mathit{then}\ \boxed{reverse1(x) = reverse(x)}^{-}$		

As in the derivation of the *reverse2* program, we may obtain from the initial goal, the definition of *reverse*, and properties of strings the new row

	$\neg(s = \Lambda)\ \wedge$ $\boxed{z = reverse\big(tail(s)\big) * head(s)}^{+}$	z

This time, however, the boxed subsentence of the goal fails to unify with the boxed subsentence of the induction hypothesis. Because the specification for *reverse1* is less general than the specification for *reverse2*, its induction hypothesis is also less general; in fact, the induction hypothesis is not general enough to unify with the desired goal.

Motivation for Generalization

In our successful derivation for *reverse1*, we have assumed that we were clever enough to first derive *reverse2*. If we were not given the specification for *reverse2*, could we, or perhaps a system, be led to discover it? In general, automatic generalization of this sort is a difficult problem. We speculate that appropriate generalizations may be discovered by observing regularities in the structure of a derivation attempt.

For example, in the attempted derivation of *reverse1* (assuming that *reverse2* has not yet been developed), we begin with the goal

$$z = reverse(s)$$

and obtain the subsentence

$$z = reverse\big(tail(s)\big) * head(s)\,.$$

If we apply the same steps to this subsentence as we did to the original goal, we obtain the subsentence

$$z = reverse\big(tail\big(tail(s)\big)\big) * head\big(tail(s)\big) * head(s)\,.$$

If we can observe the regularity in these goals, we may be inspired to construct a subprogram to satisfy instead the input-output condition

$$z = reverse(\widehat{s}) * \widehat{t}\,.$$

This is the specification for the auxiliary subprogram *reverse2*. Each of the above three subsentences is equivalent to an instance of this condition, taking $\widehat{s}$ to be s, $tail(s)$, and $tail\big(tail(s)\big)$, respectively, and $\widehat{t}$ to be Λ, $head(s)$, and $head\big(tail(s)\big) * head(s)$, respectively.

Some generalizations, however, are more difficult to motivate. For example, in the derivation of a unification algorithm (Manna and Waldinger [MW81], Nardi [Na89]), we begin with a specification

$$unify(e_1, e_2) \;\Leftarrow\; \left\{ \begin{array}{l} \textit{find } \theta \textit{ such that} \\ \quad \left[\begin{array}{l} e_1\theta = e_2\theta \;\; \wedge \\ (\forall\phi) \left[\begin{array}{l} \textit{if } \; e_1\phi = e_2\phi \\ \textit{then } \; (\exists\lambda)[\phi = \theta\lambda] \end{array} \right] \end{array} \right] \\ \qquad\qquad \vee \\ \quad \left[\begin{array}{l} (\forall\phi)\, \neg\, (e_1\phi = e_2\phi) \;\; \wedge \\ \theta = nil \end{array} \right] \end{array} \right.$$

In other words, we wish the program to return a substitution θ that is a unifier of e_1 and e_2 and that is more general than any other unifier ϕ. In the case in which e_1 and e_2 are not unifiable, the program is to return the special object *nil*, which is not a substitution.

The details of this derivation are outside the scope of this discussion. For our derivation proof to succeed, we found it necessary to add to the first disjunct of the specification the new condition $\theta\theta = \theta$, that is, θ is idempotent under composition. Nonidempotent most-general unifiers are unintuitive; for example, $\{x \leftarrow y\}$ and $\{y \leftarrow x\}$ are idempotent most-general unifiers of x and y but $\{x \leftarrow z,\, y \leftarrow z,\, z \leftarrow x\}$ is a nonidempotent most-general unifier. On the other hand, idempotence had not been studied in connection with unification, so we were surprised to require its introduction into the specification. (Idempotence has been studied independently in the work of Eder [Ed85].)

SPECIALIZED INFERENCE RULES

Progress in program synthesis depends on the development of techniques for automated deduction, both interactive and automatic. The inference rules we have introduced so far are very general: They apply to proving theorems in any theory. If we are satisfied with a more specialized system, one which is competent in a particular theory, such as the strings, we may be able to devise more powerful inference rules whose applicability is limited to that theory. Such rules may be able to achieve in a single step inferences that would otherwise require several steps.

The first benefit of this is to shorten proofs. This is a clear advantage in an interactive system, in which each step of the proof requires some effort on the part of the user. For an automatic system, a shorter proof may be an advantage if it can be found more easily. Because introducing new inference rules gives us more choices at each stage, it can actually increase the search space. Although the proof is shorter, it may be more difficult to discover.

A new inference rule may pay for itself, however, if, in addition to shortening the proof, it allows us to discard from the initial tableau some assertions that represent valid properties of the theory. We can do this only if the rule has certain completeness properties, which guarantee that in discarding the assertions we are not losing any opportunity to complete a proof. If so, the rule may reduce the number of choices at each stage and hence contract the search space.

Associative-Commutative Unification and E-Unification

One way to increase the power of an inference rule is to extend the unification algorithm to take the properties of the theory into account. For example, the associative and commutative properties of operators, such as the addition and multiplication functions in the theories of numbers or the conjunction and disjunction connectives in any logicial theory, may be incorporated into an *associative-commutative* (AC) unification algorithm (Stickel [St81]). While the ordinary unification algorithm would not be able to unify the two terms $a + (x + b)$ and $(c + a) + b$, the AC algorithm would, returning the unifier $\{x \leftarrow c\}$.

Completeness results for the algorithm have been established; that is, if the algorithm is adopted, we may discard the *associativity* property

$$(u + v) + w = u + (v + w)$$

and the *commutativity* property

$$u + v = v + u$$

from the initial tableau.

Unlike ordinary unification, which always returns a single most-general unifier, the AC unification algorithm may return a finite number of distinct unifiers. For example, for an associative-commutative function f, the result of unifying the two terms $f(a, x)$ and $f(b, y)$ can be either $\{x \leftarrow b,\ y \leftarrow a\}$ or $\{x \leftarrow f(b, u),\ y \leftarrow f(a, u)\}$, where u is a new variable.

Special unification algorithms have been devised (Siekmann [Si89]) for treating operators with various combinations of properties, including associativity, commutativity, identity, and idempotence. More general *E-unification* algorithms (e.g., Fay [Fa79]) treat operators with properties defined by a set of equations supplied by the user. Some of these algorithms produce multiple most-general unifiers or even an infinite stream of unifiers; some are not guaranteed to terminate, whether they produce an infinite stream or not.

Sorted Unification

Some unification algorithms have been devised for dealing with sort relations; these are the unary relations, such as $integer(x)$, $string(x)$, or $char(x)$, that serve to categorize our set of objects. *Sorted* unification algorithms (e.g., Schmidt-Schauss [Sc88]) allow us to provide a declaration that associates a particular sort relation with each variable and term. Thus we might declare that x is of sort *integer* and s is of sort *string*. The sorted unification algorithm will produce only substitutions of replacement pairs $x \leftarrow t$ such that x and t are of the same sort.

An advantage of using sorted unification is that we can drop from our assertions and goals all subsentences $p(t)$, where p is a sort relation. For instance, if we have declared x to be of sort *string* and y to be of sort *integer*, the sentence

$$(\forall x)(\exists y)\, q(x, y)$$

will be understood to mean

$$(\forall x)\begin{bmatrix} \textit{if } string(x) \\ \textit{then } (\exists y)[integer(y) \;\wedge\; q(x, y)] \end{bmatrix}.$$

Some assertions may disappear completely. Use of sorted unification has achieved dramatic reduction of the search space for some problems.

Extended unification algorithms may replace ordinary unification in the resolution and other inference rules of a deductive system. Where the algorithm may return multiple unifiers or fail to terminate, the control for the rule must be adapted accordingly.

Special Inference Rules

Another way to specialize a deduction system to a particular theory is to introduce entirely new inference rules. We have already seen how paramodulation (our equality rule) allows us to give special treatment to the equality relation, and thereby eliminate such axioms as transitivity and the functional-substitutivity of equality from our initial tableau. Manna and Waldinger [MW86] (and, with Stickel, [MSW91]) introduce an analogous rule for dealing with ordering relations; adopting this rule allows us to give special treatment to the ordering relation, and eliminate the transitivity and other so-called monotonicity properties of these relations. Bledsoe and Hines give special inference rules for real numbers [BH80] and set theory [Hi90].

We have seen that we can specialize a rule to a particular theory or subtheory if we have a special unification algorithm for that theory. Stickel [St85] has shown that we can also specialize a rule if we are given a procedure for determining the validity of sentences in a subtheory. The specialized rule can then be used to perform derivations in a combination of the subtheory with other theories.

For example, suppose we have two goals:

	$\boxed{z_1 \succeq b}$	z_1
	$\boxed{b \succeq a}$	b

where $\succeq$ is a total reflexive relation. The ordinary resolution rule cannot be applied because the boxed subsentences are not unifiable. If, however, we have a procedure capable of determining that, if z_1 is taken to be a, the disjunction of the instances

$$a \succeq b \ \vee \ b \succeq a$$

is valid in the total reflexive theory, then the *theory* resolution rule is able to deduce the final row

	true	*if* $a \succeq b$ *then* a *else* b

Stickel formulates completeness results that allow us to remove axioms from the initial tableau, such as the totality axiom

$$u \succeq v \ \vee \ v \succeq u\,.$$

Analogous theory extensions may be formulated for the equality rule and other inference rules. Such rules have been found to achieve sizable reductions in the search space.

DISCUSSION

There are of course many aspects of program synthesis that have not been discussed in this paper, both because of space restrictions and because many of these topics are still being developed.

We have limited ourselves to discussing the synthesis of applicative programs, which return an output but produce no side effects. Some work on the deductive synthesis of imperative programs, which may alter data structures and produce other side effects, is discussed in [MW87b]. We have also disregarded the synthesis of concurrent, real-time, and reactive programs, which may interact with their environments [e.g., PR88].

We have considered specifications only in the form of first-order input-output relations. In general, it is necessary to deal with higher-order specifications that describe properties other than input-output relations. For example, if we are constructing a pair of programs, we should be able to say that one is the inverse of the other.

We have for the most part ignored the efficiency of the programs we construct; in fact, automatically synthesized programs are often wantonly wasteful of time and space. One way of treating this is to include performance criteria as part of the program's specification; the synthesized program would then be forced to meet these criteria. Another approach is to maintain a crude performance estimate for each output entry, in a separate column. Performance estimates could be taken into account in directing the search for a program. Furthermore, once a program was constructed, the search could continue for programs with better performance estimates, based on a better algorithm or data structure, for instance.

Finally, we have concentrated on program synthesis to the exclusion of the use of deductive techniques in collaboration with other software production methods, e.g., deductive testing, debugging, verification, modification, and maintenance.

At present, progress in program synthesis is limited by the power of automated proof systems. Derivation proofs are an appealing and challenging area of application for both automatic and interactive theorem proving.

For automatic systems, program synthesis has an advantage over mathematics as an application area. To make a contribution to mathematics, a system must be able to prove theorems that a human mathematician cannot. For this reason, theorem-proving systems such as Argonne's [Mc90] have had their greatest successes in areas in which human intuition is weak, such as combinatory logic and ternary boolean algebras, so that the machine can compete on a more equal footing. For program synthesis, there is great utility in a system that can reliably be expected to prove routine and mathematically naive results, because from these results we can extract correct programs. The challenge is that many such proofs are still outside the reach of current automatic deductive technology.

To construct an interactive, rather than an automatic, program synthesis system is closer to an engineering feat today. Such a system relies on human intuition to guide the upper levels of the proof search, but itself completes the automatable details. Errors in human guidance delay the discovery of a program but never cause the system to construct an incorrect program. The challenge in designing an interactive system is to phrase the interaction in terms that the human guide can understand.

REFERENCES

[An81] P. B. Andrews, Theorem Proving via General Matings, *Journal of the ACM* 28: 2 (1981), 193–214.

[Bi83] W. Bibel, Matings in Matrices, *Communications of the ACM* 26 (1983), 844–852.

[BD77] R. M. Burstall and J. Darlington, A Transformation System for Developing Recursive Programs, *Journal of the ACM* 24: 1 (1977), 44–67.

[BH80] W. W. Bledsoe and L. Hines, Variable Elimination and Chaining in a Resolution-Based Prover for Inequalities, *Proceedings of the Fifth Conference on Automated Deduction* (1980), 281–292.

[BM79] R. S. Boyer and J S. Moore, *A Computational Logic*, Academic Press, 1979.

[BMW90] R. Burback, Z. Manna, R. Waldinger, et al, Using the Deductive Tableau System, in *MacIntosh Educational Software Collection*, Chariot Software Group, 1990.

[Con86] R. Constable et al., *Implementing Mathematics with the NuPrl Proof Development System*, Prentice-Hall, 1986.

[CH88] T. Coquand and G. Huet, The Calculus of Constructions, *Information and Control* 76: 2/3 (1988), 95–120.

[De83] N. Dershowitz, *The Evolution of Programs*, Birkhäuser, 1983.

[Dij76] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Ed85] E. Eder, Properties of Substitutions and Unifications, *Journal of Symbolic Computation* 1 (1985), 31–46.

[Fa79] M. Fay, First-Order Unification in an Equational Theory, *Proceedings of the Fourth Conference on Automated Deduction* (1979), 161–167.

[FM88] A. Felty and D. Miller, Specifying Theorem Provers in a Higher-Order Logic Programming Language, *Ninth International Conference on Automated Deduction* (1988), 61–80.

[GMW79] M. J. Gordon, A. J. Milner, C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Springer-Verlag, 1979.

[GH80] J. Guttag and J. Horning, Formal Specification as a Design Tool, *Seventh ACM Symposium on Principles of Programming Languages* (1980), 251–261.

[HK86] P. Harrison and H. Khoshnevisan, Efficient Compilation of Linear Recursive Functions into Object-Level Loops, *Proceedings of the SIGPLAN '86 Symposium on Compiler Constructions* (1986), 207–218.

[HN88] S. Hayashi and H. Nakano, *PX: A Computational Logic*, MIT Press, 1988.

[Hi90] L. Hines, Str$\dotplus$ve $\subseteq$: The Str$\dotplus$ve-based Subset Prover, *Tenth International Conference on Automated Deduction* (1990), 193–206.

[KN84] D. Kapur and P. Narendran, An Equational Approach to Theorem Proving in the First-Order Predicate Calculus, Report, General Electric Research and Development Center, Schenectady, New York, April 1984.

[Kow74] R. Kowalski, Predicate Logic as a Programming Language, *IFIP Congress 74* (1974), 569–544.

[Mc90] W. W. McCune, OTTER 2.0 Users' Guide, Mathematics and Computer Science Division, Argonne National Laboratory, 1990.

[MSW91] Z. Manna, M. Stickel, and R. Waldinger, Monotonicity Properties in Automated Deduction, in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy* (V. Lifschitz, ed.), Academic Press, 1991, 261–280.

[MW80] Z. Manna and R. Waldinger, A Deductive Approach to Program Synthesis, *ACM Transactions on Programming Languages and Systems* 2: 1 (1980), 90–121.

[MW81] Z. Manna and R. Waldinger, Deductive Synthesis of the Unification Algorithm, *Science of Computer Programming* 1 (1981), 5–48.

[MW85] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming*, Volume 1: *Deductive Reasoning*, Addison-Wesley, 1985.

[MW86] Z. Manna and R. Waldinger, Special Relations in Automated Deduction, *Journal of the ACM* 33: 1 (1986), 1–59.

[MW87a] Z. Manna and R. Waldinger, The Origin of a Binary-Search Paradigm, *Science of Computer Programming* 9 (1987), 37–83.

[MW87b] Z. Manna and R. Waldinger, The Deductive Synthesis of Imperative LISP Programs, *Sixth AAAI National Conference on Artificial Intelligence* (1987), 155–160.

[MW90] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming*, Volume 2: *Deductive Systems*, Addison-Wesley, 1990.

[M-L82] P. Martin-Löf, Constructive Mathematics and Computer Programming, *Sixth International Congress for Logic, Methodology, and Philosophy of Science* (1982), 153–175.

[Mu82] N. Murray, Completely Nonclausal Theorem Proving, *Artificial Intelligence* 18: 1 (1982), 67–85.

[Na89] D. Nardi, Formal Synthesis of a Unification Algorithm by the Deductive-Tableau Method, *Journal of Logic Programming* 7 (1989), 1–43.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's Type Theory: An Introduction*, Oxford University Press, 1990.

[Pa89] L. C. Paulson, The Foundation of a Generic Theorem Prover, *Journal of Automated Reasoning* 5: 3 (1989), 363–398.

[PR88] A. Pnueli and R. Rosner, A Framework for the Synthesis of Reactive Modules, *Proceedings of Concurrency 88* (1988), 4–17.

[Ro65] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the ACM* 12: 1 (1965), 23–41.

[Sa79] M. Sato, Towards a Mathematical Theory of Program Synthesis, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (1979), 757–762.

[Sc88] M. Schmidt-Schauss, Computational Aspects of an Order Sorted Logic with Term Declarations, SEKI Report SR–88–10, Universität Kaiserslautern, 1988.

[Si89] J. Siekmann, Unification Theory, *Journal of Symbolic Computation* 7: 3/4 (1989), 207–274.

[Sm85] D. R. Smith, Top-Down Synthesis of Divide-and-Conquer Algorithms, *Artificial Intelligence* 27: 1 (1985), 43–96.

[Sn85] N. Shankar, Checking the Proof of Gödel's Incompleteness Theorem, Institute for Computing Science, University of Texas at Austin, 1985.

[Sp83] E. Shapiro, *Algorithmic Program Debugging*, MIT Press, 1983.

[St81] M. E. Stickel, A Unification Algorithm for Associative-Commutative Functions, *Journal of the ACM* 28: 3 (1981), 423–434.

[St85] M. E. Stickel, Automated Deduction by Theory Resolution, *Journal of Automated Reasoning* 1: 4 (1985), 333–355.

[Tr89] J. Traugott, Deductive Synthesis of Sorting Programs, *Journal of Symbolic Computation* 7 (1989), 533–572.

[WR69] L. Wos and G. Robinson, Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie (editors), *Machine Intelligence 4*, American Elsevier (1969), 135–150.

[WW83] L. Wos and S. Winker, Open Questions Solved with the Assistance of AURA, in W. W. Bledsoe and D. W. Loveland (editors), *Automated Theorem Proving: After 25 Years*, American Mathematical Society (1983), 73–88.