Implementation of Intuitionistic Type Theory and Realizability Theory

Yuki Komagata and David A. Schmidt
Department of Computing and Information Sciences
Kansas State University
Manhattan, Kansas

Technical Report Number: TR-CS-95-4

Abstract

Writing correct programming code is necessary in computer system development, where complete testing is not possible. *Intuitionistic type theory* leads to a mechanical generation of correct code by using specifications. The idea is that the specification of a program is its type, and the specification can be expressed by logical statements called *well-formed formulas* (*wffs*) and therefore proved by using mathematical axioms and inference rules of logic. Then, using the correspondences *propositions are types are specifications* and *proofs are programs are values* [16], a proof can be translated into a correct programming code. The fundamental idea of *realizability theory* is that a proof can be translated into not only correct, but also minimal programming code, which contains only computational values. Based on these theories, a realizability algorithm developed by John Hatcliff defines how the translation can be done. We analyzed Hatcliff's algorithm and implemented it in a system. System development using specifications also leads to the idea of *modularization*. The *cut rule* of intuitionistic logic defines how two proofs are connected together just by looking at their specifications. This encompasses the idea of modularization and is implemented in the system. The system is then analyzed with the specification of an insertion sort program as a case study.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 Correctness of Programs

How can we write correct programming code? Writing correct programming code is necessary in the development of computer systems, especially in safety critical systems [4]. In such systems, complete testing of the program is not quite possible. Although the testing of the program is possible, in general it cannot be tested under all the possible conditions so that it can be guaranteed as correct. Two crucial concepts, types and specifications, have been used to help ensure programs are correct.

1.1.1 Types

Traditional programming languages such as PASCAL, C, and Ada use types to specify some information about data in programs. In [9], a definition of a type is given as follows:

Definition 1.1.1: A type is defined as a predicate that characterizes:

1. legal values (instances of the type), and
2. allowed operations.

Types are used in programs to detect erroneous operations or computations. For instance, types can specify that we cannot add a character to a number. A piece of information, x:A, states that x has type A. Then, it can eliminate one of two possible outcomes: either x satisfies type A or x does not. Also, types are used to compose specifications, which are defined in the next section.

1.1.2 Specifications

A specification is a more formal descriptor than a type. In [9], a definition of a specification is given as follows:

Definition 1.1.2:

> A specification of a program component is a statement of the function computed by the program component. The purpose of using specifications includes:
> • correctness of the system at the first steps of the development, and
> • a high productivity in the development effort.

A specification must state necessary conditions for all items that are computed by the program component [9]. Thus, the user of the program relies on the specification for the use of the program, and the programmer relies on the specification for the construction of the program [15]. When specifications are used in the system development, we have to ensure the following:

- the specification says what you want to say, and
- the programming code matches the specification [4].

The first statement is usually related to the communication problem between the user (client of the system) and the programmer. The second statement concerns the problem of writing correct programming code, which is what we are interested in.

System development using specifications can separate the design from the implementation so that the design is independent of the implementation. Usually, in system development, a system is divided into several smaller components called *modules*. A *module* is a grouping of related operations and variables that are shared by the module operations [9]. Each module in a system has a specification, and it can be designed and implemented separately. Each specification states relationships to other modules in the system so that after it is implemented, all modules can be connected to form one system and work as one system. In this system development, modules are connected only by looking at the specifications, and not the actual code.

One of the languages used to write specifications of computer programs is *predicate calculus*. Specifications are expressed by *predicate calculus formulas*.

Example:

By using a predicate calculus formula, we can write a specification of a factorial:

$\forall x \in nat. \exists y \in nat.\ y = x!$

where x and y are natural numbers represented as "nat."

The specification states that "for all natural numbers x, there exists a natural number y, which equals the factorial of x."

Proofs are needed to determine whether the conditions stated by the specifications are true. Predicate calculus is used in the two ways: for describing the conditions and for reasoning [15]. In the next section, we will study briefly the connections of logic proofs and programs.

1.2 Logic Proofs and Computer Programs

The predicate calculus we use as a specification language is intuitionistic predicate logic. In the following section, informal descriptions of *intuitionistic logic* and traditional *classical logic* are provided.

1.2.1 Intuitionistic Logic and Classical Logic

It was Brouwer who originally introduced *intuitionistic logic* [2]. He is the founder of the intuitionistic approach in mathematics. The existing results concerning intuitionistic formal systems can be found in [4,17]. *Intuitionistic logic* has been developed based on constructive mathematics whereas *classical logic* is based on traditional nonconstructive mathematics. The underlying idea of intuitionism is that a proof of a mathematical claim should provide a construction (algorithm) of the number that satisfies the claim. Because of the constructive (computational) ideas, which are related to and can be applied to the areas of computer science, intuitionistic logic has received much attention, and much work has been done in its development.

Example:

We consider a mathematical statement:

$$\forall x \in nat. \exists y \in nat.\ y > x$$

which states that "for all natural numbers x, there exists a natural number y such that y is greater than x." In constructive mathematics, the heart of the proof of this statement is the number, x+1, because for all natural numbers x, 1 added to x is greater than x itself, that is, x+1>x. Further, the function, $\lambda x \in nat.x+1$, can even be considered the *construction* or the program that results from the proof of the proposition. In intuitionistic logic, these ideas are formalized, and we describe how we implemented them in the next section.

1.3 Implementation Overview

This section gives an overview of the M.S. project. The example of the previous section suggests that a proof of a proposition can be converted into a program, which satisfies the proposition. Since the proof guarantees the specification (proposition), the program which results from the proof of the proposition also guarantees the specification. In other words, it is a correct program. With these ideas in mind, we took the following steps in order to implement the ideas of intuitionistic logic. A brief explanation is given about each step.

• Martin-Löf's Type Theory

We studied Martin-Löf's type theory, which is one formalization of intuitionistic logic. Martin-Löf states that the specification of a program is its type, and his interpretation leads to the correspondences *propositions are types* and *proofs are programs* [13]. Martin-Löf's type theory shows how to convert a proof of a proposition into a program that satisfies the proposition. This is shown in the following example.

Example:

The proof of the claim:

$\forall x \in nat.\ \exists y \in nat.\ y>x$

in Martin-Löf's type theory generates the program:

$\lambda x \in nat.$ (x+1, "proof that x+1>x holds").

The program maps a number, x, to an output pair consisting of x+1 and an explanation why x+1 is correct, that is, why x+1>x holds. Martin-Löf's type theory is explained in Chapter 2.

• Realizability Theory

We then studied realizability theory, which states how a proof can be mechanically translated into a minimal program. A program developed by Martin-Löf's type theory contains both "computation parts" and "proof parts," and only the computation parts are extracted by the realizability theory translation. The idea is to represent the computation (also called construction) parts by natural numbers (or functions of natural numbers) [4]. The following example shows the realizability theory translation.

Example:

When a realizability algorithm is applied to the above example, we obtain the more concise program: $\lambda x \in nat.$ x+1 - the proof part is discarded. The type of the program - $\forall x \in nat.\ \exists y \in nat.\ y>x$ - remembers what the discarded proof part actually proved.

• Hatcliff's Realizability Algorithm

We implemented a realizability algorithm written by John Hatcliff. The algorithm shows how to extract computer programs from proofs in intuitionistic logic, and is explained in detail in Chapter 2.

• Implementation of Hatcliff's Realizability Algorithm

After studying Hatcliff's algorithm, we implemented it to build a system to translate a logical proof into programming code, which has only computation parts.

• Implementation of Cut Rule

With the idea of modules used in system development as mentioned earlier, we studied the cut rule used in intuitionistic logic, which shows how to connect two proofs together by their specifications. We then implemented the cut rule to do linking of programs. This let us link together proved-correct programs into a proved-correct system. The cut rule is explained in Chapter 2.

• Case Study

Once the system was implemented, we tested it on an insertion sort example. We first proved the specification for an insert function, "realized" it to a program, and then we proved the specification for an insertion sort by using the insert function as an assumption, and "realized" it to another program. We then linked the two programs together by using the cut rule. The realized programs are compared with the programs written in *Standard ML*.

1.4 Structure of this Report

The steps described in the previous section are further explained in the following chapters of this report. The first of five chapters in this report is an introduction. Chapter 2 presents background materials by briefly introducing the formal logic system, including both propositional logic and predicate logic. The discussion of type theory and realizability theory then leads to the correspondence between formal logic proofs and programming languages. In Chapter 3, the implementation issues used to build the system are discussed. Chapter 4 presents a case study with test examples of the system, where an insertion sort program is developed. Finally, Chapter 5 concludes the report and suggests possible future work on the project.

Chapter 2
Intuitionistic Logic and Type Theory

2.1 Formal Logic System

Traditionally, it is said that logic is an art or a study of reasoning [3]. A formal logic system provides a language of logic to express the concepts and process of reasoning. It consists of:

- a syntax definition of the sentences of the logic,
- a set of axioms and inference rules to prove the correct sentences, and
- a notion of proof.

The sentences of the logic are called *well-formed formulas (wffs)*, and they can be formed by the set of fundamental symbols defined in the language of the logic.

Two formal logic systems, propositional logic and predicate logic, are studied in the following sections.

2.2 Propositional Logic

Propositional logic deals only with propositions, that is, declarative statements which are simply true or false. The language of propositional logic consists of:

- proposition symbols P, Q, R, ...
- connectives $\wedge$, $\vee$, $\supset$, $\neg$, $\perp$
- auxiliary symbols $\vdash$, (,)

The syntax definition is given in the following BNF:

$S \in$ Sequent
$\Gamma \in$ Context
$\phi \in$ Proposition
$P \in$ Proposition-symbol

$S ::= \Gamma \vdash \phi$
$\Gamma ::= \phi 1, \phi 2, ..., \phi n \quad n>=0$
$\phi ::= P \mid \phi 1 \wedge \phi 2 \mid \phi 1 \vee \phi 2 \mid \phi 1 \supset \phi 2 \mid \neg \phi 1 \mid \perp$

The proposition symbols are called atomic propositions since they cannot be decomposed further. Atomic propositions can form more complex propositions with the connectives. They are defined recursively as ϕ in the above syntax definition. Each proposition symbol stands for a statement, which has a truth value. For example, if P stands for "John is tall," and Q stands for "Mary likes John," then, $P \wedge Q$ stands for "John is tall and Mary likes John."

2.2.1 Sequent

As shown in the above syntax definition, the sequent, $\Gamma \vdash \phi$, has three components: Γ is called the *context* or *assumptions*, ϕ is called the *conclusion*, and they are separated by the symbol, $\vdash$, which is called a *turnstile.* A sequent, $\Gamma \vdash \phi$, states that a proposition ϕ is derived from assumptions Γ. In a proof of $\Gamma \vdash \phi$, ϕ is derived by applying inference rules to given assumptions (initial assumptions), Γ. Thus, Γ can be treated as *axioms*. The context may be empty, that is, Γ may contain no propositions, and the sequent is the form, $\vdash \phi$, meaning that ϕ is inferred by using no assumptions. In that case, ϕ is called a *theorem*.

2.2.2 Connectives

Connectives form a more complex proposition from simpler ones. Connectives carry traditional names such as:

$\wedge$ - conjunction (and),
$\vee$ - disjunction (or),
$\supset$ - implication (if ... then ...),
$\neg$ - negation (not), and
$\perp$ - falsum (false).

The first three connectives are binary connectives, and the negation $\neg$ is a unary connective. Because we are using *intuitionistic logic*, rather than *classical logic*, $\neg\phi$ is in fact an abbreviation for $\phi \supset \perp$. In traditional classical logic, $\neg\phi$ is treated differently based on the idea of *excluded middle*, where for any proposition ϕ, $\phi \vee \neg\phi$ holds. On the other hand, intuitionistic logic is based on the idea that a proof of a

proposition must be computed or constructed, therefore $\phi \vee \neg\phi$ is not true for all ϕ[1]. The symbol, $\perp$, is an unusual connective, which does not connect anything. For that reason, it is called a *logical constant.*

2.2.3 Assumption

An assumption is the simplest inference rule of all. It states that we can infer a proposition which appears in the context. The rule is represented as a sequent with the same proposition in the context and in the conclusion as shown below:

Assumption: $\Gamma \vdash \phi$, if $\phi \in \Gamma$

2.2.4 Inference Rules

Each connective has two kinds of inference rules: an *introduction* rule and an *elimination* rule. An introduction rule builds a proposition with a specific connective, and an elimination rule removes a specific connective from a proposition. For each rule, the sequents above the line are called *premises* or *antecedents*, and the one below the line is called the *conclusion* or *succedent.* Each rule is stated below.

Conjunction Introduction:

$$(\wedge I)\ \frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \wedge \phi_2}$$

Conjunction Elimination:

$$(\wedge E)\ \frac{\Gamma \vdash \phi_1 \wedge \phi_2 \quad \Gamma, q1{:}\phi_1, q2{:}\phi_2 \vdash \phi}{\Gamma \vdash \phi}$$

Disjunction Introduction (Left):

$$(\vee IL)\ \frac{\Gamma \vdash \phi_1}{\Gamma \vdash \phi_1 \vee \phi_2}$$

[1]But it is true, however, that all mathematical formulas for arithmetic such as $(x>0)\vee\neg(x>0)$ can be proved [18].

Disjunction Introduction (Right):

$$(\ \vee_{IR}\ )\ \frac{\Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \vee \phi_2}$$

Disjunction Elimination:

$$(\ \vee_{E}\ )\ \frac{\Gamma \vdash \phi_1 \vee \phi_2 \quad \Gamma, q1{:}\phi_1 \vdash \phi \quad \Gamma, q2{:}\phi_2 \vdash \phi}{\Gamma \vdash \phi}$$

Implication Introduction:

$$(\ \supset_{I}\ )\ \frac{\Gamma, x{:}\phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \supset \phi_2}$$

Implication Elimination:

$$(\ \supset_{E}\ )\ \frac{\Gamma \vdash \phi_1 \supset \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}$$

Falsum Elimination:

$$(\ \perp_{E}\ )\ \frac{\Gamma \vdash \perp}{\Gamma \vdash \phi}$$

Note that the introduction rule for negation, $\neg$, is indeed the implication introduction rule:

$$\frac{\Gamma, x{:}\phi \vdash \perp}{\Gamma \vdash \neg\phi}$$

The elimination rule for negation, $\neg$, is indeed the implication elimination rule:

$$\frac{\Gamma \vdash \neg\phi \quad \Gamma \vdash \phi}{\Gamma \vdash \perp}$$

since $\neg\phi$ is the same as $\phi \supset \perp$.

Example:

Let's consider a proof of P⊃Q, P ⊢ P∧Q. The proposition P∧Q is derived from the two assumptions, P⊃Q and P by using the inference rules described in the previous section. A proof can be represented in the form of a tree whose root is a desired proposition, ϕ, whose leaves are axioms, and whose internal nodes are applications of inference rules. Such a tree is called a *natural deduction tree*. Figure 2.2 shows the natural deduction tree for the proof of P⊃Q, P ⊢ P∧Q.

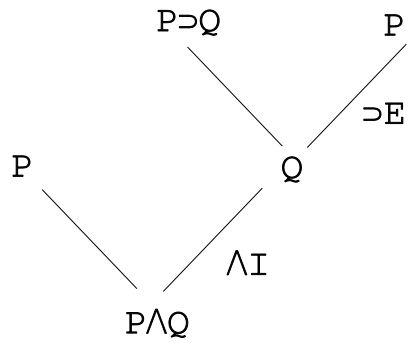


Figure 2.2 Natural Deduction Tree for P⊃Q, P ⊢ P∧Q

2.3 Predicate Logic

As described in the previous section, propositional logic deals with propositions. Unfortunately, it is not sufficient to express a simple statement such as "for all natural numbers x, x+1 is greater than x." Propositional logic talks about relations of propositions, but cannot reason about individual objects such as numbers. We wish to talk about all individuals, and dually, some individuals as in, "there exists a number x which is greater than zero and less than 2." In order to do so, the new connectives, ∀ and ∃ are introduced. The resulting logic is called predicate logic, and it is considered an extension of propositional logic. The language of predicate logic consists of:

- function symbols +, *, -, 0, 1, ...
- predicate symbols <, >, =, ...
- identifiers x, y, z, a, b, ...
- connectives ∀, ∃, ∧, ∨, ⊃, ¬, ⊥
- auxiliary symbols ., ⊢, (,)

The syntax definition is given in the following BNF:

$S \in$ Sequent $f \in$ Function-symbol
$\Gamma \in$ Context $P \in$ Predicate-symbol
$\phi \in$ Proposition $x \in$ Identifier
$t \in$ Term

$S ::= \Gamma \vdash \phi$
$\Gamma ::= \phi 1, \phi 2, ..., \phi n \quad n>=0$
$\phi ::= \forall x.\phi \mid \exists x.\phi \mid \phi 1 \wedge \phi 2 \mid \phi 1 \vee \phi 2 \mid \phi 1 \supset \phi 2 \mid \neg \phi 1 \mid \perp \mid$
$P(t1, t2, ..., tn), n>=0$
$t ::= x \mid f(t1, t2, ..., tn) \; n>=0$

Predicate symbols represent *predicates*, which may have zero or more arguments (also called parameters). A *predicate* can be considered a relation or some property over its arguments if it has any. The arguments of predicates are called *terms* or *expressions*, which are either identifiers or function symbols, where a function may have zero or more arguments. The difference between a predicate and a function is that a function can be calculated and results in a constant, whereas a predicate represents some relation or property. Since only terms are considered to be the arguments of predicates, what we are using is called *first order logic*. There is also *higher order logic*, where the arguments of predicates can also be predicates.

2.3.1 Quantifiers

The new connectives, $\forall$ and $\exists$, are called *quantifiers*. A *universal quantifier*, $\forall$, talks about all individuals, and an *existential quantifier*, $\exists$, talks about an existence of certain individuals. A universal quantifier $\forall$ reads "for all" and an existential quantifier $\exists$ reads "there exists." With these quantifiers, we can express the above arguments, "for all natural numbers x, x+1 is greater than x" as $\forall x \in nat.x+1>x$, and, "there exists a number x which is greater than zero and less than 2" as $\exists x \in nat.x>0 \wedge x<2$.

2.3.2 Free and Bound Variables

The quantifiers bind identifiers in a specific scope, and notions of *free* and *bound* variables are introduced. The set of free variables in a proposition ϕ, FV(ϕ), is

defined inductively as follows:

Definition 2.3.2: A set of free variables FV(ϕ) is:

$FV(\forall x.\phi 1) = FV(\phi 1) - \{x\} = FV(\exists x.\phi 1)$

$FV(\phi 1 \wedge \phi 2) = FV(\phi 1) \cup FV(\phi 2) = FV(\phi 1 \supset \phi 2) = FV(\phi 1 \vee \phi 2)$

$FV(\neg\phi) = FV(\phi)$

$FV(\perp) = \{\}$

$FV(P(t1, t2, ..., tn)) = FV(t1) \cup FV(t2) \cup ... \cup FV(tn)$

$FV(x) = \{x\}$

The set of free variables in a context is: $FV(\Gamma) = \{x | x \in FV(\phi), \phi \in \Gamma\}$.

An identifier is free if it does not fall into the scope of some quantifier; otherwise it is bound. As mentioned in [3], the same variable may occur both free and bound in a proposition. The free variables play an important role in substitution, which is described in the next section.

Example:

$FV(\forall x.P(x) \wedge Q(x)) = \{\}$

$FV((\forall x.P(x)) \wedge Q(x)) = \{x\}$

In the first example, there are no free variables. The second example shows the case where variable x occurs both free and bound.

2.3.3 Substitution

Quantifiers bind variables in propositions indicating the properties of the variables. When we have actual instances of the variables, we substitute them for the variables. For example, consider a proposition, $\forall x.P(x)$. If we have an actual instance of the variable x, say t, then we can talk about the property P of t, that is, P(t) by substituting t for x. Here, the substitution is nothing but the process of replacement, which has the notation: $\phi[t/x]$, that is, term t substitutes for all free occurrences of variable x in proposition ϕ. The substitution is defined as follows:

Definition 2.3.3: A substitution, $\phi[t/x]$, is defined as:

$(\forall y.\phi 1)[t/x] = \forall y.\phi 1[t/x]$ if $x <> y$ and $y \notin FV(t)$

$(\forall y.\phi 1)[t/x] = \forall z.\phi 1[z/y][t/x]$ if $x <> y$ and $y \in FV(t)$

$(\forall x.\phi 1)[t/x] = \forall x.\phi 1$

$(\exists y.\phi 1)[t/x] = \exists y.\phi 1[t/x]$ if $x <> y$ and $y \notin FV(t)$

$(\exists y.\phi 1)[t/x] = \exists z.\phi 1[z/y][t/x]$ if $x <> y$ and $y \in FV(t)$

$(\exists x.\phi 1)[t/x] = \exists x.\phi 1$

$(\phi 1 \wedge \phi 2)[t/x] = \phi 1[t/x] \wedge \phi 2[t/x]$

$(\phi 1 \vee \phi 2)[t/x] = \phi 1[t/x] \vee \phi 2[t/x]$

$(\phi 1 \supset \phi 2)[t/x] = \phi 1[t/x] \supset \phi 2[t/x]$

$(\neg \phi 1)[t/x] = \neg \phi 1[t/x]$

$\perp[t/x] = \perp$

$P(t1, t2, ..., tn)[t/x] = P(t1[t/x], t2[t/x], ..., tn[t/x])$

When we substitute one variable for another, we need to be aware of *capturing*, that is, a free variable falls into some scope of another variable with the same name by substitution and becomes bound. This capture of free variables under substitution is considered undesirable [8]. In order to avoid capturing, one possible solution is to change the bound variable names in the proposition prior to the substitution whenever needed. This is seen in the third and sixth rules in Definition 2.3.3. Examples are provided below to show the capturing issues.

Example:

$((\forall x.P(x)) \wedge Q(x))[1/x] = (\forall x.P(x)) \wedge Q(1)$

$(\forall x.x+y)[x/y] = \forall x.x+x$

$(\forall x.x+y)[x/y] = \forall x.x+y[z/x][x/y] = \forall z.z+x$

The first example demonstrates the substitution for only free occurrences of x. The second example shows the problem of capturing after substituting x for y. The free variable y, which is now substituted for x, is trapped in the scope of the universal quantifier and becomes a bound variable. The third example shows a possible solution of the capturing, where the bound variable x has changed its name to z prior to the substitution.

2.3.4 Inference Rules

The inference rules of predicate logic are the ones from propositional logic together with the introduction and elimination rules for each quantifier listed below.

Universal Introduction

$$(\forall I)\ \frac{\Gamma \vdash \phi(x)}{\Gamma \vdash \forall x.\phi(x)}\ \textit{if } x \notin FV(\Gamma)$$

Universal Elimination

$$(\forall E)\ \frac{\Gamma \vdash \forall x.\phi(x)}{\Gamma \vdash \phi(t)}$$

Existential Introduction

$$(\exists I)\ \frac{\Gamma \vdash \phi(t)}{\Gamma \vdash \exists x.\phi(x)}$$

Existential Elimination

$$(\exists E)\ \frac{\Gamma \vdash \exists x.\phi 1(x) \qquad \Gamma,\ \phi 1(a) \vdash \phi 2}{\Gamma \vdash \phi 2}\ \textit{if } a \notin FV(\Gamma) \textit{ and } a \notin FV(\phi 2)$$

Example:

We will demonstrate a rather familiar proof of $\exists x.\forall y.P(x,y) \vdash \forall y.\exists x.P(x,y)$, that is, a proof to change the order of two quantifiers. As in the example proof in Figure 2.2, the proof is represented by the natural deduction tree. All four inference rules introduced in this section are used in the proof. The numbering placed next to the inference rule and the proposition indicate the scope of the local assumption.

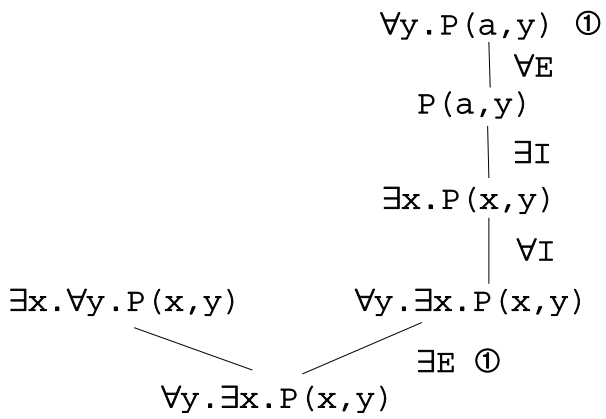


Figure 2.3 Natural Deduction Tree for ∃x.∀y.P(x,y) ⊢ ∀y.∃x.P(x,y)

2.4 Conversion of Proofs to Programs

As mentioned earlier, the logic that we are studying is intuitionistic logic. Its constructive idea motivates a development of correct programming code. In the following sections, we give informal discussions of several aspects of intuitionistic logic related to the development of our system, which generates correct computer programs. Because proofs guarantee the correctness of what the specifications say, the fundamental idea lies in the conversion of logical proofs to correct programming code. First, the natural deduction tree from Figure 2.2 is analyzed.

2.4.1 Linearization of Proofs into Lambda Calculus

In Figure 2.2, we have introduced a tree representation of proofs called *a natural deduction tree*. It shows the consequence of inference rules used in the proof, and each node can be thought of as a subproof of the proof, which can be inferred by the inference rule in its arc to its parent node. Thus, the tree precisely tells us how each proof step is done in order. We can write the tree in a linear form starting from the root down to the leaves, and it still represents each proof step in order as in the natural deduction tree. This process is called *linearization*, and it is done mechanically by taking each inference rule and its direct child nodes as its arguments from the root of the tree to its leaves, and the corresponding linearized form reads

from left to right. We use the approaches given in [16], and some examples are provided below to show how the linearization works.

Example:

The proof in Figure 2.2 can be linearized into the following form:

$(\wedge I\ (a2)\ (\supset E\ (a1\ a2)))$

The first inference rule used in the proof is conjunction introduction, which has two nodes for the subproofs for P and Q. The subtree with implication elimination is reformatted similarly. Assumptions $P \supset Q$ and P are named a1 and a2, respectively. Another example, the proof in Figure 2.3, can be linearized as shown below. The "assumed" proof tree for the assumption, $\exists x.\forall y.P(x,y)$, is named a1.

$(\exists E\ a1\ (a \in V,\ b \in (\forall y \in V.P(a,y)))\ (\forall I\ (y \in V)\ (\exists I_{\exists x.P(x,y)}\ a\ (\forall E\ b\ y))))$

The existential elimination rule has three parts: a proof of an existential quantified formula, local assumptions, and a proof inferred from the local assumptions with other assumptions and axioms. The local assumptions of the existential elimination rule are named a and b and used in the rest of the proof.

Our goal is to make proofs look like programs. After linearizing proof trees, we can take another step to replace the inference rule names with corresponding programming symbols. The result is *lambda calculus expressions*. Again, we use the approaches in [16] to reformat a proof into a computation, and they are described below. The lambda calculus expressions, e, e1, and e2, correspond to the reformatted linear forms of the natural deduction trees.

- $(\wedge I\ e1\ e2)$ is formatted as: (e1, e2), that is, an ordered pair
- $(\wedge E\ e1\ (p \in \phi 1,\ q \in \phi 2)\ e2)$ is formatted as: let (p,q)=e1 in e2
- $(\supset I\ (x \in \phi)\ e)$ is formatted as: $\lambda x \in \phi.e$, that is, a lambda abstraction
- $(\supset E\ e1\ e2)$ is formatted as: $e1 \bullet e2$,
  that is, an application of a lambda abstraction
- $(\vee IL_{\phi 1 \vee \phi 2}\ e)$ is formatted as: $inl_{\phi 1 \vee \phi 2}\ e$,
  that is, a tagged element of disjoint union
- $(\vee IR_{\phi 1 \vee \phi 2}\ e)$ is formatted as: $inr_{\phi 1 \vee \phi 2}\ e$,
  that is, a tagged element of disjoint union

- ($\vee$E e1 (x$\in$1) e2 (y$\in$2) e3) is formatted as:

  case e1 of isl(x$\in\phi$1).e2 ‖ isr(y$\in\phi$2).e3, that is, a case statement
- ($\perp$E$_\phi$ e) is formatted as: abort$_\phi$ e
- ($\forall$I (x$\in$V) e) is formatted as: λx$\in$V.e, that is, a lambda abstraction
- ($\forall$E e t) is formatted as: e•t, that is, an application of a lambda abstraction
- ($\exists$I$_{\exists x.\phi(x)}$ t e) is formatted as: (t, e), that is, an ordered pair
- ($\exists$E e1 (a$\in$V, b$\in\phi$(a)) e2) is formatted as: open e1 as (a$\in$V, b$\in\phi$(a)).e2

This reformatting process is shown in the example below with actual linear forms of proofs. The reformatting matches Heyting's interpretation described in the next section.

Example:

The linear form of the proof of P$\supset$Q, P $\vdash$ P$\wedge$Q in the above example can be reformatted as the following lambda calculus expression:

(a2, (a1•a2))

As described above, the implication introduction is replaced with a pair, and the implication elimination is replaced with an application. Another example, the linear form of the proof of $\exists$x.$\forall$y.P(x,y) $\vdash$ $\forall$y.$\exists$x.P(x,y) is reformatted as the following lambda calculus expression:

open a1 as (a$\in$V, b$\in$($\forall$y$\in$V.P(a,y))).λy$\in$V.(a, b•y)

2.4.2 Heyting's Interpretation

Heyting's interpretation is very original. His interpretation is that the meaning of propositions are the proofs of the propositions, not the truth values of the propositions. They are stated as follows:

- a proof of ϕ1$\wedge\phi$2

  = a pair, consisting of a proof of ϕ1 and a proof of ϕ2
- a proof of ϕ1$\supset\phi$2

  = a function that transforms proofs of ϕ1 into proofs of ϕ2
- a proof of ϕ1$\vee\phi$2

  = a proof of ϕ1, labeled by *inl*, or a proof of ϕ2, labeled by *inr*
- a proof of $\forall$x$\in$V.ϕ(x)

= a function that transforms an individual, $x \in V$, into a proof of $\phi(x)$

- a proof of $\exists x \in V.\phi(x)$

  = a pair, consisting of an individual, $t \in V$, called a *witness*,
  and a proof for $\phi(t)$

- a proof of $\perp$ does not exist - this is related to the negation, $\neg\phi$, which is equivalent to $\phi \supset \perp$, where $\perp$ is a statement that does not have a possible proof

- a proof of an arithmetic statement - we assume that we know what a proof is

  Example: For an arithmetic statement, 2+1=3, "pencil and paper calculation serves as a proof of the statement [7]." Also, each number itself serves as a proof of itself, that is, for 0, 1, 2, ..., each number is a proof of each individual natural number.

2.4.3 Martin-Löf's Intuitionistic Type Theory

The ideas of converting logic proofs into computer programs described in this report are in fact based on Martin-Löf's intuitionistic type theory [13,14]. The formulae-as-types (propositions-as-sets) interpretation, on which intuitionistic type theory is based, leads to important correspondences: *propositions are types* and *proofs are programs* [13]. In [14], he then provides further explanations of the close connection between programs and constructive mathematics together with its logic, that is, intuitionistic logic, which explains why the intuitionistic type theory may be considered equal to a programming language. The presentation in Section 2.4.1 is based upon Martin-Löf's type theory.

2.4.4 Realizability Theory

The original idea of realizability was introduced by Kleene in 1945 [11,12]. The motivation is that proofs are a mixture of computation parts and proof parts, and we want to extract only the computation parts. This is the main idea of realizability theory, and it is developed by Turner [19] and Troelstra and van Dalen [18]. Under the development of intuitionistic logic, the fact that "$\exists$ and $\vee$ carry most of constructiveness" is evidenced, and it is also stated that "constructive existence is

found with ∃ and ∨" [18]. Therefore, formulas with ∃ or ∨ are called *positive* and extracted as the computation parts. The following section gives explanations of the version of realizability that we use in our system.

2.5 Hatcliff's Realizability Algorithm

Based on the ideas of intuitionism and realizability described in the previous sections, John Hatcliff has developed an algorithm to convert a logical proof to a computer program. Hatcliff accepts Heyting's interpretation and Kleene's realizability theory and adopts the formalization of the evidence model provided by Martin-Löf's type theory. His notion of realizability theory and algorithm is described in the following sections.

2.5.1 Polarity

A polarity is either positive (+) or negative (-), which indicates a positive or negative formula as defined by Turner [19]. In [5], the definition of negative formulas, which is similar to Turner's, is provided as follows:

Definition 2.5.1: Negative formulas:

> A formula is said to be *negative* if it does not contain ∃ and ∨; otherwise it is said to be *positive*.

Thus, in Hatcliff's words, a negative formula is "∃, ∨-free." The definition is based on the idea of intuitionistic logic, where only the existential quantifier, ∃, and the disjunction connective, ∨, have computational values. In Hatcliff's algorithm, the negative parts are thrown away, and only positive parts remain in the converted programs. As a result, the programs contain only computational values, and for that reason, they are in fact terms (expressions) after the conversion.

Polarity of Propositions

As described above, the fundamental idea is that a proposition is positive if it contains either ∨, the disjunction connective, or ∃, the existential quantifier; otherwise it is negative. Therefore, the polarity can be calculated mechanically based on the connectives and quantifiers of the proposition, and is done recursively as:

- $\phi 1 \wedge \phi 2$ is positive if $\phi 1$ or $\phi 2$ is positive; otherwise negative,
- $\phi 1 \vee \phi 2$ is always positive,
- $\phi 1 \supset \phi 2$ is positive if $\phi 2$ is positive; otherwise negative,
- $\neg\phi$ is always negative ($\neg\phi$ is an abbreviation of $\phi \supset \perp$),
- $\forall x{:}\tau.\phi$ is positive if ϕ is positive; otherwise negative,
- $\exists x{:}\tau.\phi$ is always positive,
- P(e1, e2, ..., en), n>=0, is negative, and
- τ is negative,

  where P is a predicate symbol, e1, e2, ..., en compose an expression (term) list, and τ is a data type.

Note that a positive formula may contain negative subformulas, and vice versa.

Example:

According to the above list, we can mechanically calculate the polarity of a proposition. Suppose that P is a positive primitive proposition, and Q is a negative primitive proposition. A proposition is positive if it contains either $\exists$ or $\vee$; otherwise, it is negative. Thus, P might stand for: $\exists x \in$nat. x>0, and Q might stand for: $\forall y \in$nat. y+1>y. We wish to calculate the polarity of a proposition, $P \wedge Q$. To do so, we need to know the polarities of both P and Q of the conjunction. We can put the polarity of a proposition, which is either positive + or negative -, next to the proposition, that is:

"P is positive" can be expressed as "P+"

"Q is negative" is expressed as "Q-"

Since P is positive, $P \wedge Q$ is calculated as positive, which can be expressed as:

$(P+ \wedge Q-)+$ or simply $(P \wedge Q)+$

How about the polarity of another proposition, $P \supset Q$? We have $P+ \supset Q-$, and for the implication, the polarity depends on the polarity of the proposition on the right hand side of the connective, that is, the polarity of Q. Therefore, we can express it as:

$(P+ \supset Q-)-$ or simply $(P \supset Q)-$

Polarity of Proofs

The polarity of a proof is calculated based on its type, i.e. the proposition proved. We

can determine whether a proof of $\Gamma \vdash \phi$ is positive or negative by the polarity of proposition ϕ. Therefore, a proof of $\Gamma \vdash \phi$ is said to be positive if the proposition ϕ is positive.

Example:

Suppose that we have proofs, p and q, for the propositions, P and Q, respectively, which are described in the previous example. The polarity of a proof is the polarity of the proposition proved, and we can say that the proof is positive if its proposition is positive. Since proposition P is positive, proof p is positive. Similarly, since proposition Q is negative, proof q is negative. As propositions, proofs can be expressed with polarities as p+ and q-. Propositions with connectives work the same way. The proof of proposition $P \wedge Q$, say p1, is positive since the proposition, $P \wedge Q$, is positive. Similarly, the proof of proposition $P \supset Q$, say p2, is negative since the proposition, $P \supset Q$, is negative. They can be expressed as: p1+ and p2-.

2.5.2 Hatcliff's Algorithm

The important observations in Hatcliff's algorithm include:

- there is a difference between propositions and data types, and
- quantifiers can be used only with data types.

Based on these observations, propositions and data types are treated differently, especially in the calculation of polarities.

A proof is translated into a program depending on polarities of each subproof in the proof and the inference rule used in the proof. This mechanism is defined in Hatcliff's algorithm of realizability theory shown in Figure 2.5.2. In the algorithm, p is a proof, q is a variable associated with a proposition, x is a variable associated with a data type, ϕ is a proposition, and τ is a data type. The notation, $[\![\ ]\!]$, is the translation function $[\![\ ]\!]$: proof -> program, that is, a proof is translated into the programming code on the right hand side of the arrow. As stated in the examples in the previous section, the plus + and minus - symbols shown next to proofs such as p+ and variables such as q1- indicate polarities of the components.

⟦p⁻⟧ ⇒ *:1
⟦∧-I p1⁺ p2⁺⟧ ⇒ (⟦p1⁺⟧, ⟦p2⁺⟧)
⟦∧-I p1⁺ p2⁻⟧ ⇒ ⟦p1⁺⟧
⟦∧-I p1⁻ p2⁺⟧ ⇒ ⟦p2⁺⟧
⟦∧-E (q1⁺, q2⁺)=p1 in p2⁺⟧ ⇒ let (q1, q2)=⟦p1⟧ in ⟦p2⁺⟧
⟦∧-E (q1⁺, q2⁻)=p1 in p2⁺⟧ ⇒ let q1=⟦p1⟧ in ⟦p2+⟧
⟦∧-E (q1⁻, q2⁺)=p1 in p2⁺⟧ ⇒ let q2=⟦p1⟧ in ⟦p2+⟧
⟦∧-E (q1⁻, q2⁻)=p1 in p2⁺⟧ ⇒ ⟦p2⁺⟧
⟦∨-I1 p⟧ ⇒ inl ⟦p⟧
⟦∨-I2 p⟧ ⇒ inr ⟦p⟧
⟦∨-E p of q1.p1⁺ ‖ q2.p2⁺⟧ ⇒ case ⟦p⟧ of (inl q1).⟦p1⁺⟧ ‖ (inr q2).⟦p2⁺⟧
⟦⊃-I q∈ϕ⁺.p⁺⟧ ⇒ λq∈⟦ϕ⁺⟧.⟦p⁺⟧
⟦⊃-I q∈ϕ⁻.p⁺⟧ ⇒ ⟦p⁺⟧
⟦⊃-E p1⁺⊃⁺ p2⟧ ⇒ ⟦p1⁺⟧•⟦p2⁺⟧
⟦⊃-E p1⁻⊃⁺ p2⟧ ⇒ ⟦p⁺⟧
⟦∀I x∈τ.p⁺⟧ ⇒ λx∈⟦τ⟧.⟦p⁺⟧
⟦∀E p⁺ t⟧ ⇒ ⟦p⁺⟧•⟦t⟧
⟦∃I t p⁺⟧ ⇒ (⟦t⟧, ⟦p⁺⟧)
⟦∃I t p⁻⟧ ⇒ ⟦t⟧
⟦∃E (x, q⁺)=p1⁺ in p2⁺⟧ ⇒ let (x, q)=⟦p1⁺⟧ in ⟦p2⁺⟧
⟦∃E (x, q⁻)=p1⁻in p2⁺⟧ ⇒ let q=⟦p1⟧ in ⟦p2+⟧

Figure 2.5.2 Hatcliff's Algorithm for Realizability Theory

Example:

Before explaining the translation process of the algorithm, let's consider a few examples. First, suppose that we have a sequent, ⊢ P⊃(Q⊃(P∧Q)), where proposition P is positive, P+, and proposition Q is negative, Q-, as in the examples of the previous section. In order to apply Hatcliff's algorithm in Figure 2.5.2, there are several steps to follow:

1. First, we calculate the polarities of propositions. The polarities of each component is calculated, and it is expressed as follows:

   (P+∧Q-)+

   (Q-⊃(P∧Q)+)+

   (P+⊃(Q⊃(P∧Q))+)+

   Note that the overall polarity of the component is the right most polarity. The overall polarity of the conclusion, P⊃(Q⊃(P∧Q)), is positive.

2. We now need to prove the proposition, and write the proof in a linear form with the syntax matched to the algorithm. The proof is as follows:

   $(\supset\text{-I } p{\in}P.\ (\supset\text{-I } q{\in}Q.\ (\wedge\text{-I } p\ q)))$

   Note that p and q are hypothetical proofs for P and Q respectively.

3. We then calculate and attach the polarities of subproofs needed for the translation. The polarities of proofs are the polarities of the propositions, which are calculated in the first step. We calculate the polarities from the innermost proofs, that is, the proof by conjunction introduction. By looking at the polarities of the propositions shown in the step 1, we attach them to the proof in the step 2:

   $(\wedge\text{-I } p+\ q-)+$

   $(\supset\text{-I } q{\in}Q-.\ (\wedge\text{-I } p\ q)+)+$

   $(\supset\text{-I } p{\in}P+.\ (\supset\text{-I } q{\in}Q.\ (\wedge\text{-I } p\ q))+)+$

4. Finally, we can translate the above proof into a programming code using the algorithm. We check from the outermost translation by finding which case in the algorithm applies depending on the inference rule and the polarities of components.

   $[\![\supset\text{-I } p{\in}P+.\ (\supset\text{-I } q{\in}Q.\ (\wedge\text{-I } p\ q))+]\!] \Rightarrow \lambda p{\in}[\![P+]\!].[\![\supset\text{-I } q{\in}Q.\ (\wedge\text{-I } p\ q)]\!]$

   $\Rightarrow \lambda p{\in}[\![P+]\!].[\![\supset\text{-I } q{\in}Q-.\ (\wedge\text{-I } p\ q)+]\!]$

   $\Rightarrow \lambda p{\in}[\![P+]\!][\![\wedge\text{-I } p+\ q-]\!]$

   $\Rightarrow \lambda p{\in}P.p$

   Note that only the positive components are translated further. The translation of any assumption including local ones such as p and q results in the variable itself referring to a hypothetical proof. The translated programming code for a proof of $\vdash P\supset(Q\supset(P\wedge Q))$ is a lambda abstraction: $\lambda p{\in}p.p$

Now we consider another example proof of

$\forall m{\in}nat.m{\neq}0 \supset (m\text{-}1)+1=m \vdash \forall x{\in}nat.(x{\neq}o)\supset\exists x{\in}nat.y+1=x$

We take the same steps as in the above example.

1. To calculate the polarities of propositions, we can decompose the assumption and the conclusion into several parts, and then calculate from the innermost parts as follows:

Assumption: $(m \neq 0)-$ $((m-1)+1=m)-$

$(m \neq 0 \supset (m-1)+1=m))-$

$(\forall m \in nat.(m \neq 0 \supset (m-1)+1=m))-$

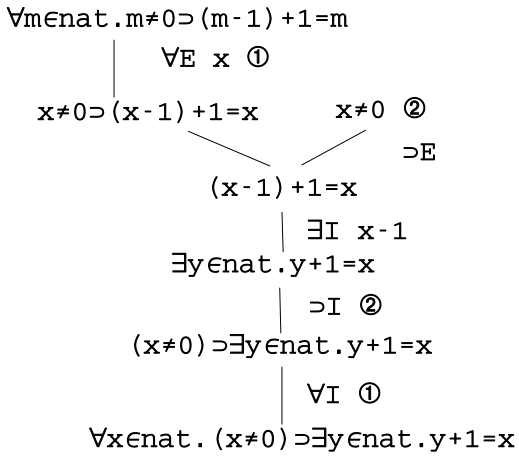
Conclusion: $(y+1=x)-$ $(\exists x \in nat.y+1=x)+$

$((x \neq o)- \supset (\exists x \in nat.y+1=x)+)+$

$(\forall x \in nat.(x \neq o) \supset (\exists x \in nat.y+1=x))+$

Therefore, the polarity of the assumption is negative and the polarity of the conclusion is positive.

2. Now we prove the sequent. The proof of the sequent is shown as a natural deduction tree, then it is formatted to a linear form.



The linear form of the tree is:

$\forall I\ (x \in nat)\ (\supset\text{-}I\ (p \in (x \neq o))\ (\exists I\ x\text{-}1\ (\supset\text{-}E\ (\forall E\ a\ x)\ p)))$

where the assumption is called a.

3. We then show the polarities of the proof from the innermost components similar to step 1: $(\forall E\ a-\ x)-$

$(\supset\text{-}E\ (\forall E\ a\ x)\text{-}\supset\text{-}\ p)-$

$(\exists I\ x\text{-}1\ (\supset\text{-}E\ (\forall E\ a\ x)\ p)\text{-})+$

$(\supset\text{-}I\ (p \in (x \neq o))\text{-}\ (\exists I\ x\text{-}1\ (\supset\text{-}E\ (\forall E\ a\ x)\ p))+)+$

$(\forall I\ (x \in nat)\ (\supset\text{-}I\ (p \in (x \neq o))\ (\exists I\ x\text{-}1\ (\supset\text{-}E\ (\forall E\ a\ x)\ p))))+$

The polarity of the subproof $(\forall E\ a\ x)$ is shown with an implication connective as $-\supset-$. This is because it proves a proposition with an implication, and it is used in the proof proved by implication elimination rule.

4. The translation of the above proof is done according to the algorithm. Each subproof is translated as follows starting from the outermost one:

$[\![\forall I\ (x \in nat)\ (\supset\text{-}I\ (p \in (x \neq o))\ (\exists I\ x\text{-}1\ (\supset\text{-}E\ (\forall E\ a\ x)\ p)))+]\!]$

$\Rightarrow \lambda x \in nat.[\![\supset\text{-}I\ (p \in (x \neq 0))\text{-}\ (\exists I\ x\text{-}1\ (\supset\text{-}E\ (\forall E\ a\ x)\ p))+]\!]$

$\Rightarrow \lambda x \in nat.[\![\exists I\ x\text{-}1\ (\supset\text{-}E\ (\forall E\ a\ x)\ p)\text{-}]\!]$

$\Rightarrow \lambda x \in nat.[\![x\text{-}1]\!] \Rightarrow \lambda x \in nat.x\text{-}1$

Therefore, the translation results in a lambda abstraction for any natural number x, which subtracts one from x.

The other proof is for $\exists x.\forall y.P(x,y) \vdash \forall y.\exists x.P(x,y)$, and the assumption is called a1. The linear form of the proof is as follows:

$(\exists E\ (a, b)=(a1)\ in\ (\forall I\ y \in nat\ (\exists I\ a\ (\forall E\ b\ y))))$

We attach the polarities to each components:

$(\exists E\ (a, b\text{-})=(a1)\ in\ (\forall I\ y \in nat.\ (\exists I\ a\ (\forall E\ b\text{-}\ y)\text{-})+)+)+$

The translation is as follows:

$[\![\exists E\ (a, b\text{-})=(a1)\ in\ (\forall I\ y \in nat.\ (\exists I\ a\ (\forall E\ b\text{-}\ y)))+]\!]$

$\Rightarrow$ let $b=[\![a1]\!]$ in $[\![\forall I\ y \in nat.\ (\exists I\ a\ (\forall E\ b\ y))+]\!]$

$\Rightarrow$ let $b=a1$ in $\lambda y \in [\![nat]\!].[\![\exists I\ a\ (\forall E\ b\text{-}\ y)\text{-}]\!]$

$\Rightarrow$ let $b=a1$ in $\lambda y \in nat.[\![a]\!] \Rightarrow$ let $b=a1$ in $\lambda y \in nat.a$

2.5.3 Explanation of Hatcliff's Algorithm

The translation process of each inference rule is explained below in order. Subproofs are represented as p, p1, and p2, e is an expression or a term, t is a data type, and x is a variable.

Empty Program: $[\![p^-]\!] \Rightarrow *:1$

Any negative proof is translated into an empty program represented by an asterisk, *, because it does not have any computational information. An empty program is of type 1, which indicates that there is only one programming code generated from the translation of any negative proof, that is, an empty program.

Example:

By using Hatcliff's algorithm shown in Figure 2.5.2, the proofs in Figures 2.2 and 2.3 can be translated into programs. The first proof is for P⊃Q, P ⊢ P∧Q. Assume that primitive propositions P and Q are both negative, and we call the assumptions a1 for P⊃Q, and a2 for P. The linear form of the proof is shown as: (∧I a2 (⊃E a1 a2))
We attach the polarities to each component:

(∧I a2- (⊃E a1-⊃- a2)-)-

The translation is done from the left to right as follows:

⟦∧I a2- (⊃E a1 a2)-⟧ ⇒ *:1

The resulting programming code is an empty program.

Conjunction Introduction: ⟦∧-I p1 p2⟧

⟦∧-I p1+ p2+⟧ ⇒ (⟦p1+⟧, ⟦p2+⟧)

⟦∧-I p1+ p2−⟧ ⇒ ⟦p1+⟧

⟦∧-I p1− p2+⟧ ⇒ ⟦p2+⟧

If both p1 and p2 are positive, the translation results in a pair consisting of two translated programs as in the first case; otherwise, only the positive subproof is translated further.

Example:

A short example is provided to demonstrate one of the above translation cases. Let's consider a proof of P, Q ⊢ P∧Q, where P is positive and Q is negative. The assumptions are called a1 for P and a2 for Q. The sequent with the polarities is:

P+, Q- ⊢ (P+ ∧ Q-)+

The proof of the sequent is:

(∧-I a1 a2)

Then the polarities are attached to p and q:

(∧-I a1+ a2-)+

Since P is positive and Q is negative, the polarity of the entire proof is positive. This is the second case of the conjunction introduction in the algorithm, and the translation is as follows:

⟦∧-I a1+ a2-⟧ ⇒ ⟦a1+⟧ ⇒ a1

that is, the positive assumption a1 of type P.

Conjunction Elimination: ⟦∧-E (q1, q2)=p1 in p2⟧

⟦∧-E (q1$^+$, q2$^+$)=p1 in p2$^+$⟧ ⇒ let (q1, q2)=⟦p1⟧ in ⟦p2$^+$⟧

⟦∧-E (q1$^+$, q2$^-$)=p1 in p2$^+$⟧ ⇒ let q1=⟦p1⟧ in ⟦p2+⟧

⟦∧-E (q1$^-$, q2$^+$)=p1 in p2$^+$⟧ ⇒ let q2=⟦p1⟧ in ⟦p2+⟧

⟦∧-E (q1$^-$, q2$^-$)=p1 in p2$^+$⟧ ⇒ ⟦p2$^+$⟧

The two components of conjunction in p1 are called q1 and q2, and they can be used in p2. This is used in the translation where only the positive components of p1 are substituted in p2. In the first case p1 substitutes for two variables, and the second and third cases, p1 substitutes only for the variable associated with the positive proposition. No substitution occurs in the last case since both components of p1 are negative.

Example:

Consider a proof of P∧Q ⊢ P, where P is positive and Q is negative propositions. The assumption P∧Q is called a1. The sequent with the polarities is:

P+ ∧ Q- ⊢ P+

The proof of the sequent is:

(∧-E (p, q)=a1 in p)

where p and q are the names for each component of the conjunction, that is, for P and Q respectively. Then we attach the polarities to the proof as follows:

(∧-E (p+,q-)=a1 in p+)+

This is the second case of the conjunction elimination rule in the algorithm. The translation is as follows:

⟦∧-E (p+,q-)=a1 in p+⟧ ⇒ let p=⟦a1⟧ in ⟦p+⟧ ⇒ let p=a1 in p

The programming code is a substitution, where the assumption a1 substitutes for all free occurrences of p in p, which results in just the assumption a1. The let notation, however, states the substitution, but the actual substitution does not occur.

Disjunction Introduction Left: ⟦∨-I1 p⟧

⟦∨-I1 p⟧ ⇒ inl(⟦p⟧)

The disjunction introduction rules are used to prove the proposition of the form $\phi 1 \vee \phi 2$, and the proof is always positive. The proof p is translated, and the result is "tagged" as *inl*.

Disjunction Introduction Right: ⟦∨-I2 p⟧

⟦∨-I2 p⟧ ⇒ inr(⟦p⟧)

The translation is done in the same way as the disjunction introduction left rule described above except that the tag is *inr*.

Disjunction Elimination: ⟦v-E p of q1.p1 ‖ q2.p2⟧

⟦∨-E p of q1.p1+ ‖ q2.p2+⟧ ⇒ case ⟦p⟧ of (inl q1).⟦p1+⟧ ‖ (inr q2).⟦p2+⟧

If the subproofs p1 and p2 are positive, the proof is translated into a case statement, where the first case q1 is the translated program of p1 and the second case q2 is the translated program of p2. Here, p is a proof of a proposition of the form $\phi 1 \vee \phi 2$, where each component is called q1 of type $\phi 1$ and q2 of type $\phi 2$. The three subproofs, p, p1, and p2, are translated further.

Example:

Consider a proof of $P \supset R, Q \supset R \vdash (P \vee Q) \supset R$, where P and R are positive and Q is negative. The assumptions are called a1 for $P \supset R$ and a2 for $Q \supset R$. The sequent with the polarities is shown below:

$P+ \supset R+, Q- \supset R+ \vdash ((P+ \vee Q-)+ \supset R+)+$

The proof of the sequent is as follows:

(⊃-I a∈P∨Q (∨-E a of p.(⊃-E a1 p) ‖ q.(⊃-E a2 q)))

where p and q are the local assumptions for P and Q respectively in the disjunction elimination. The proof with the polarities are shown below:

(⊃-I a∈(P∨Q)- (∨-E a of p.(⊃-E a1+⊃+ p)+ ‖ q.(⊃-E a2-⊃+ q)+)+)+

The translation of the above proof is shown as follows:

$[\![\supset\text{-I } a\in(P\vee Q)\text{- } (\vee\text{-E } a \text{ of } p.(\supset\text{-E } a1\ p) \parallel q.(\supset\text{-E } a2\ q)))^+]\!]$

$\Rightarrow [\![\vee\text{-E } a \text{ of } p.(\supset\text{-E } a1\ p)^+ \parallel q.(\supset\text{-E } a2\ q)^+]\!]$

$\Rightarrow \text{case } [\![a]\!] \text{ of } (\text{inl } p).[\![\supset\text{-E } a1^+\supset^+ p]\!] \parallel (\text{inr } q).[\![\supset\text{-E } a2\text{-}\supset^+ q]\!]$

$\Rightarrow \text{case } a \text{ of } (\text{inl } p).[\![a1^+]\!]\bullet[\![p^+]\!] \parallel (\text{inr } q).[\![a2^+]\!]$

$\Rightarrow \text{case } a \text{ of } (\text{inl } p).a1\bullet p \parallel (\text{inr } q).a2$

that is, the case statement on the condition of $P\vee Q$.

Implication Introduction: $[\![\supset\text{-I } q\in\phi.p]\!]$

$[\![\supset\text{-I } q\in\phi^+.p^+]\!] \Rightarrow \lambda q.[\![\phi^+]\!].[\![p^+]\!]$

$[\![\supset\text{-I } q\in\phi^-.p^+]\!] \Rightarrow [\![p^+]\!]$

The implication introduction corresponds to a lambda abstraction. If both p and ϕ are positive, the translation results in a lambda abstraction as in the first case; otherwise, if p is positive, but not ϕ, then it is solely the translation of p as in the second case.

Example:

Recall the first example proof of $\vdash P\supset(Q\supset(P\wedge Q))$ given in this section. Both cases of implication introduction are used in the example.

Implication Elimination: $[\![\supset\text{-E } p1\ p2]\!]$

$[\![\supset\text{-E } p1^+\supset^+ p2]\!] \Rightarrow [\![p1^+]\!]\bullet[\![p2^+]\!]$

$[\![\supset\text{-E } p1^-\supset^+ p2]\!] \Rightarrow [\![p^+]\!]$

The implication elimination rule corresponds to an application. Here, p1 is a proof of $\phi1\supset\phi2$, and p2 is a proof of $\phi1$. If p1 and p2 are positive as in the first case, both are translated and form an application; otherwise if p2 is negative, it is solely the translation of p1 as in the second case.

Universal Introduction: $[\![\forall\text{I } x\in\tau.p]\!]$

$[\![\forall\text{I } x\in\tau.p^+]\!] \Rightarrow \lambda x.[\![\tau]\!].[\![p^+]\!]$

Similar to the implication introduction rule, the universal introduction rule corresponds to a lambda abstraction except that the variable x is a data type rather than a proposition. Therefore, if the subproof p is positive, the translation results in a lambda

abstraction, τ and p are translated further.

Example:

Recall the second example proof of:

$\forall m \in nat.\ m \neq 0 \supset (m-1)+1=m \vdash \forall x \in nat.\ (x \neq 0) \supset \exists y \in nat.\ y+1=x$

in this section. The above case of universal introduction is used in the example.

Universal Elimination: $[\![\forall E\ p\ t]\!]$

$[\![\forall E\ p^{+}\ t]\!] \Rightarrow [\![p^{+}]\!] \bullet [\![t]\!]$

Similar to the implication elimination rule, the universal elimination rule corresponds to an application. If the subproof p is positive, p and t are translated further, and the translation results in an application of the translated t to the translated p. Note that t is an expression.

Example:

Recall the second example proof of:

$\forall m \in nat.\ m \neq 0 \supset (m-1)+1=m \vdash \forall x \in nat.\ (x \neq 0) \supset \exists y \in nat.\ y+1=x$

in this section. The other case of universal introduction, that is, the case where the translation results in an empty program, is shown in the example.

Existential Introduction: $[\![\exists I\ t\ p]\!]$

$[\![\exists I\ t\ p^{+}]\!] \Rightarrow ([\![t]\!], [\![p^{+}]\!])$

$[\![\exists I\ t\ p^{-}]\!] \Rightarrow [\![t]\!]$

Similar to the conjunction introduction rule, the existential introduction rule corresponds to an ordered pair consisting of an instance and its proof. The existential introduction is used to prove the proposition of the form $\exists x \in \tau.\phi$, and the proof is always positive. Thus, if the subproof p is positive, it is translated and paired with the translated expression t as in the first case. Otherwise, if the subproof is negative, the translation of the proof is solely the translation of the expression t as in the second case.

Example:

Recall the second example proof of:

$\forall m \in nat.\ m \neq 0 \supset (m-1)+1=m \vdash \forall x \in nat.\ (x \neq 0) \supset \exists y \in nat.\ y+1=x$

in this section. The second case of existential introduction is used in the example.

Existential Elimination: $[\![\exists E\ (x, q)=p1 \text{ in } p2]\!]$

$[\![\exists E\ (x, q^+)=p1^+ \text{ in } p2^+]\!] \Rightarrow \text{let } (x, q)=[\![p1^+]\!] \text{ in } [\![p2^+]\!]$

$[\![\exists E\ (x, q^-)=p1^- \text{ in } p2^+]\!] \Rightarrow \text{let } x=[\![p1]\!] \text{ in } [\![p2+]\!]$

Similar to the conjunction elimination rule, the existential elimination rule corresponds to substitution. Here, p1 proves a proposition of the form $\exists x \in \tau.\phi$, and it is always positive. If both p2 and q are positive, the subproof p1 is translated, and the translation is the substitution of p1 for both x and q in the translated p2 as in the first case; otherwise if only the subproof p2 is positive, then it is the substitution of p1 for x in p2 as in the second case.

Example:

Consider a proof of $\exists x \in nat.P(x) \vdash \exists y \in nat.P(y)$, which changes the name of the quantified variable name. Assume that proposition P is negative. The assumption is called a1, and the polarities can be attached as follows:

$(\exists x \in nat.P(x)^-)^+ \vdash (\exists y \in nat.P(y)^-)^+$

The proof is given in a linear form as following:

$(\exists E\ (a,b)=a1 \text{ in } (\exists I\ a\ b))$

where the two local assumptions for existential elimination are called $a \in nat$ and $b \in P(a)$.

The proof tree with polarities can be expressed as:

$(\exists E\ (a,b^-)=a1^- \text{ in } (\exists I\ a\ b^-)^+)^+$

Then, the translation is made by using the second case of existential elimination as follows:

$[\![\exists E\ (a,b^-)=a1^- \text{ in } (\exists I\ a\ b^-)^+]\!] \Rightarrow \text{let } a=[\![a1]\!] \text{ in } [\![\exists I\ a\ b^-]\!]$

$\Rightarrow \text{let } a=a1 \text{ in } [\![a]\!] \Rightarrow \text{let } a=a1 \text{ in } a$

The programming code is a substitution of the assumption a1 for all free occurrences of a in a, that is, just the assumption a1.

There are some missing cases in Hatcliff's algorithm, such as ⟦∧-I p1– p2-⟧, but this is not by accident. Hatcliff optimizes the cases, which result in an empty program, *, and puts them together as in the case: ⟦p–⟧ ⇒ *:1. Thus, if a proof does not match to any case in the algorithm according to the inference rule used in it, then the translation results in an empty program, *.

2.6 Linking of Programs with Specifications

In the previous section, logical proofs are translated into computer programs by using Hatcliff's algorithm. The programs are guaranteed correct since they are translated from correct logical proofs. We wish to link the programs just by looking at the specifications (types), but not the programs. By using the specifications in system development, there are several advantages including an isolation of the design from the implementation described in Chapter 1. We can link the realized programming code, which is developed from the specification. This linking process is implemented by the *cut rule*.

2.6.1 Cut Rule

The cut rule shown in Figure 2.6.1 states how two specifications, that is, propositions, are linked into one. The cut rule states that if we have a proof, say p1, whose conclusion is used as an assumption of another proof, p2, then we can link two proofs together by reducing the proof steps. The resulting proof is p2, where the assumption proved by p1 is removed from the context and that assumption used in p2 is substituted by the actual proof, p1. This substitution process follows our discussion in Section 2.3.3. Since the two proofs are correct proofs, the "linked" proof is also guaranteed to be correct.

$$\frac{\Gamma \vdash \phi 1 \qquad \Delta, \phi 1 \vdash \phi 2}{\Gamma, \Delta \vdash \phi 2}$$

Figure 2.6.1 Cut Rule

Example:

The cut rule links two programs together by using their specifications. Let's consider two programs, p1 and p2, shown below:

program p1: $\forall x \in nat.\ x+1>x \vdash \forall x \in nat.\ \exists y \in nat.\ y>x$

program p2: $\forall x \in nat.\ \exists y \in nat.\ y>x,\ \forall m \in nat.\ m>0 \supset m \neq 0 \vdash \exists n \in nat.\ n \neq 0$

Since the conclusion of p1 is used by p2 as an assumption, we can link them together by using the cut rule. To do so, first we translate p1 and p2 into programs by using Hatcliff's algorithm described in the previous chapter, and then link them together by using the specifications. The assumption of p1 is called a1 and the assumptions of p2 are called a2 and a3 from the left to right. Thus, p1 proves the assumption a2. The sequent with polarities are expressed as follows:

p1: $(\forall x \in nat.\ (x+1>x)\text{-})\text{-} \vdash (\forall x \in nat.\ (\exists y \in nat.(y>x)\text{-})+)+$

p2: $(\forall x \in nat.\ (\exists y \in nat.\ (y>x)\text{-})+)+$,

$(\forall m \in nat.\ ((m>0)\text{-} \supset (m \neq 0)\text{-})\text{-} \vdash (\exists n \in nat.\ (n \neq 0)\text{-})+$

Both p1 and p2 prove positive propositions because of the existential quantifier $\exists$. The assumptions a1 and a3 are negative, but a2 is positive. The proofs are shown in linear forms below:

p1: $(\forall I\ x \in nat\ (\exists I\ x+1\ (\forall E\ a1\ x)))$

p2: $(\exists E\ (a,b)=(\forall E\ a2\ 0)\ in\ (\exists I\ a\ (\supset\text{-}E\ (\forall E\ a3\ a)\ b)))$

The proofs with polarities are shown as follows:

p1: $(\forall E\ a1\text{-}\ x)\text{-}$

$(\exists I\ x+1\ (\forall E\ a1\ x)\text{-})+$

$(\forall I\ x \in nat\ (\exists I\ x+1\ (\forall E\ a1\ x))+)+$

p2: $(\forall E\ a3\text{-}\ a)\text{-}$

$(\supset\text{-}E\ (\forall E\ a3\ a)\text{-}\supset\text{-}\ b)\text{-}$

$(\exists I\ a\ (\supset\text{-}E\ (\forall E\ a3\ a)\ b)\text{-})+$

$(\exists E\ (a,b\text{-})=(\forall E\ a2+\ 0)\text{-}\ in\ (\exists I\ a\ (\supset\text{-}E\ (\forall E\ a3\ a)\ b))+)+$

Then, we can translate p1 and p2 as follows:

p1: ⟦∀I x∈nat (∃I x+1 (∀E a1 x))+⟧
⇒ λx∈⟦nat⟧.⟦∃I x+1 (∀E a1 x)-⟧ ⇒ λx∈nat.⟦x+1⟧ ⇒ λx∈nat. x+1

p2: ⟦∃E (a,b-)=(∀E a2+ 0)- in (∃I a (⊃-E (∀E a3 a) b))+⟧
⇒ let a=⟦∀E a2+ 0⟧ in ⟦∃I a (⊃-E (∀E a3 a) b)-⟧
⇒ let a=⟦a2+⟧•⟦0⟧ in ⟦a⟧ ⇒ let a=a2•0 in a

The program for p1 is a lambda abstraction of number x that adds 1 to x, and the program for p2 is a substitution of an application of a2 to 0 for all free occurrences of a in a, that is, just an application of the assumption a2 to 0. Now we have two programs each with a specification. Now we apply the cut rule in Figure 2.6.1. The assumption a2 used to prove p2 is now proved by p1. Thus, when p1 and p2 are linked together, a2 is removed from the context of p2, which results in the following specification:

a1, a3 ⊢ ∃n∈nat.n≠0

Now we know that program p2 uses proof p1 for assumption a2, and a2 is replaced by p1 in p2, that is, the linked program is the form:

let a2=p1 in p2

which is a substitution of p1 for all free occurrences of a2 in p2. The actual linked program is expressed as:

let a2=λx∈nat. x+1 in let a=a2•0 in a

Chapter 3
Implementation of the System

This chapter describes the implementation of our system. It has a sequence of tasks to be accomplished, and the implementation is done in the order listed below:

1. define a logic system to be used,
2. implement a proof checker for the logic system,
3. implement a realizer to calculate and attach polarities to proofs,
4. implement a translator to translate proofs into programs according to Hatcliff's algorithm for realizability theory, and
5. implement a linker by using the cut rule and the β-rule.

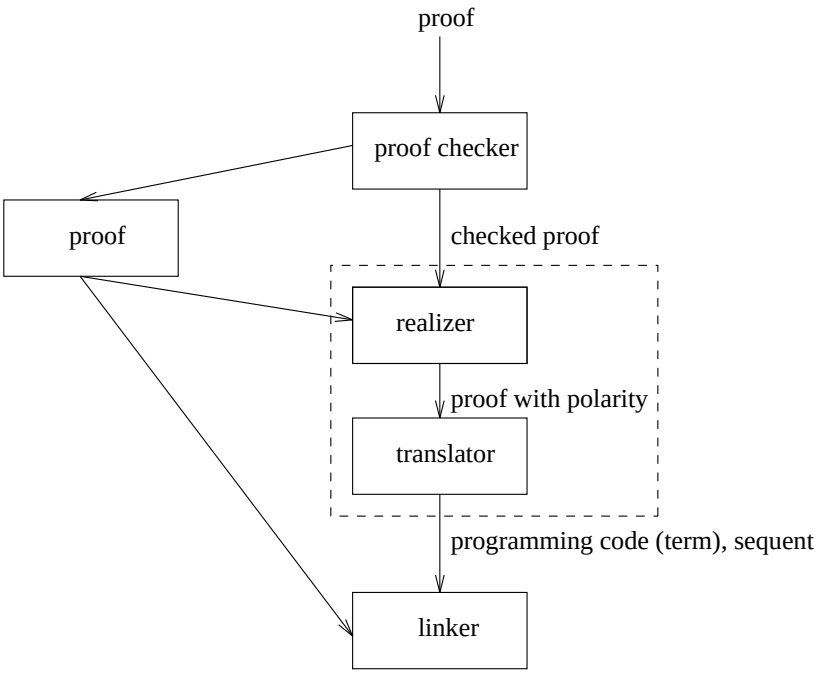


Figure 3.1 Block Diagram of the System

The system is written in *Standard ML* using *SML/NJ (0.93)* as a run-time system. Each step is explained in the following sections. The system can be divided into three main modules as shown in Figure 3.1. Files used in our system with brief descriptions are as follows:

- lex.sml - lexical analyzer (scanner),
- parse.sml - syntax analyzer (parser),
- constraints.sml - semantic analyzer (constraints checker),
- proof.sml - defines and loads input/output data types,
- realize.sml - realizer - calculates and attaches polarities for proofs,
- translate.sml - translator - translates proofs to programs,
- link.sml - linker - links two programs together,
- make.sml - make file to load the system, and
- test.sml - defines functions to run and test the system.

3.1 Proof Checker - the Front End

The proof checking system written by Dr. David Schmidt is the front end of our system. It consists of three files: lex.sml, parse.sml, and constraints.sml. The proof checker takes an input file name, which contains a proof, parses the proof to build a parse tree, checks if it is correct, and produces the proof tree as an output. As described in the previous chapter, because of the correspondence between programming languages and formal logical proofs, usual compiler techniques of programming languages described in [1] can be applied to formal logical proofs.

3.1.1 Input Proof Language

The input to the proof checker is an input file name which contains a proof. The syntax of the proof language that we use for the system is defined in BNF as shown in Figure 3.1.1. The only nonblank separators in the grammar are: , . ( ) and :.

```
m ::=  { predicate w { : tau { , tau }* }}*  pf
pf ::=  { d {,} }*  { body }
p ::=  ( pf )
body ::=  embed id | assume b { , b }* in pf end | use w | phi0 by r | _ by r
b ::=  id : phi0
d ::=  line w = pf
phi0 ::=  ! id : tau . phi0  |  ? id : tau . phi0  |  phi1
phi1 ::=  phi2 { -> phi2 }*
phi2 ::=  phi3 { v phi3 }*
phi3 ::=  phi4 { & phi4 }*
phi4 ::=  ( phi0 )  |  - phi4  |  tau  |  w { (  e { , e }* ) }
w  ::=  STRING
id  ::=  STRING beginning with lower case alphabetic
e  ::=  nil  |  cons e1 e2 |  NUM  |  succ e  |  ( e ) |  { embed } id
tau ::=  nat  |  listof tau  | ( tau )
r  ::=  id |  &I  p1  p2  |  &E  p1  p2  |  vIL  p  |  vIR  p  |  vE  p1  p2  p3
        |  ->I  p  |  ->E  p1  p2  |  !I p  |  !E  p  e  |  ?I  e  p  |  ?E  p1  p2
        |  listE  p1  p2  p3  |  law  |  arithmetic
```

Figure 3.1.1 Grammar for Input Proof Language

In the grammar, **m** is the overall proof structure, which consists of the list of predicates used in a proof and the proof, **pf**. A **pf** consists of the optional list of abstractions for proofs, **d**, and the body of the proof, **body**. A **body** is either a reference to an assumption, **embed id**, which coerces the assumption into a proof; a subproof with local assumptions in an assume block, **assume ... end**; an abstraction invocation, **use w**; or a proposition followed by an inference rule used in the proof, **phi0 by r**, where the proposition may be omitted, **_ by r**, in some cases. The last two forms are described further in the following section. A binder, **b**, used in an assume block, indicates an association between an identifier, **id**, and a proposition, **phi0**. This is used to define each assumption in a context. An abstraction, **d**, is the form **line w = pf**, and it indicates that a proof **pf** can be referred to as a name **w** by the abstraction invocation of the form **use w**. The abstraction feature is described further below.

Simplification Features in the Grammar

There are some convenient "short-cut" features available in the grammar. The general form of the proof body can be written as **phi0 by r**, where **phi0** is a proposition

indicating what a proof is trying to prove, and **r** is an inference rule used in the proof. This form can be simplified by omitting the proposition **phi0** and replacing with an underscore _ as an abbreviated form **_ by r**. The proof checker calculates the target proposition by looking at the inference rule, **r**. The abbreviated form can be used for all inference rules except the disjunction introduction rule and the existential introduction rule; because of their characteristics, the proof checker cannot calculate the target propositions. Another simplification feature is the abstraction used for a proof, which allows us to name it by using the form **line w = pf**, then the corresponding abstraction invocation is in the form **use w**, where **w** is a string.

Example:

There is more than one way to write a proof in the grammar. Here, we will illustrate an example proof of P⊃Q, P ⊢ P∧Q from Figure 2.2 in several different ways with or without using the simplification features described above. Figure 3.1.1a shows the proof written without any simplification features. In Figure 3.1.1a, the first two lines with the keyword **predicate** list all the primitive propositions used in the proof. Each **embed** constructor refers to an assumption a or b, and is used to utilize the assumption in the proof. Notice how the **assume_end block** is used to introduce assumptions, here a:P->Q, and b:P, and specify the scopes.

```
predicate P
predicate Q

assume a : P -> Q , b : P in
   P & Q by &I ( embed b ) ( Q by ->E ( embed a ) ( embed b ) )
end
```

Figure 3.1.1a Input Proof of P⊃Q, P ⊢ P∧Q

Figure 3.1.1b shows the same proof with simplification features. The abstraction feature is used by naming the subproof for Q as 1 and the main proof for P∧Q as 2, then the proof is just an abstraction invocation for the main proof, 2. Then, the abbreviation _ feature is used by omitting the proposition, which indicates what is being proved.

```
predicate P
predicate Q

assume a : P -> Q , b : P in
   line 1 = _ by ->E ( embed a ) ( embed b )
   line 2 = _ by &I ( embed b ) ( use 1 )
   use 2
end
```

Figure 3.1.1b Input Proof of P⊃Q, P ⊢ P∧Q With Simplification Features

By using these features, subproofs can be organized, and writing and repetitions are reduced especially for large proofs. The output proofs of the proof checker for the above two input proofs are identical.

A proposition, **phi**, is recursively defined as either a proposition with a predicate logic connective, a data type, **tau**, or a user defined predicate with zero or more expressions for its parameter, **w (e0, ..., en)**. Due to the limitations of the characters on the keyboard, logical connectives are mapped to some other symbols with ASCII characters. Each mapping is listed as follows:

- universal quantifier ∀ maps to !
- existential quantifier ∃ maps to ?
- implication ⊃ maps to ->
- disjunction ∨ maps to v
- conjunction ∧ maps to &
- negation ¬ maps to -

At the moment, we will only consider natural numbers and lists of natural numbers. Thus, a **tau** is either a natural number, **nat**, or a list of natural numbers, **listof nat**, which is possibly nested as **listof listof nat**. Similarly, an expression, **e**, is either a natural number or a list. If an **e** is a number, then it can be a number itself such as 0 or 45, or a successor of a number, **succ e**. If an **e** is a list, then it can be an empty list, **nil**, or non-empty list, **cons e1 e2**. If an identifier has an appropriate data type

defined by **tau**, then an **e** can also be represented as an identifier, **id**. Any proposition, expression, and data type may be parenthesized.

An inference rule, **r**, is used in a body of a proof, **body**, and it consists of an assumption invocation, **id**, and inference rules of predicate logic, as well as the list elimination rule, **listE p1 p2 p3**, and two other rules, **law** and **arithmetic**, which are described in the following section. Proofs used in inference rules, **r**, must be parenthesized as **p**, that is, **(pf)**. The only difference between two proofs, **pf** and **p**, is a pair of parentheses, that is, **p** is a proof surrounded by a pair of parentheses whereas **pf** is not. This distinction is made to simplify the process of parsing. All proofs that appear in **r** must be **p**, a parenthesized proof, and no other proofs in the grammar may have surrounding parentheses.

Proof by Law and by Arithmetic

In addition to inference rules for logical connectives, the last two rules, **law** and **arithmetic**, are also provided. Both rules are used as "a wild card," which can prove anything, so that actual proof steps are omitted. The purpose of the rule **law** is to utilize already-known-to-be-true propositions, such as mathematical axioms and theorems, without actually proving them. Similarly, the purpose of the rule **arithmetic** is to omit proofs by assuming that the target propositions can be proved and taking a chance that they might not be able to be proved. Thus, the **arithmetic** rule should not be used for proofs that contains important parts to be used for translation later on. In other words, the rule **arithmetic** cannot be used for positive proofs. The notion of positive and negative proofs will be discussed in a later section.

Example:

>(x,0) or - >(x,0) by law `(x^i+y^i=z^i for i≥3) by arithmetic`

The first example shows the arithmetic law of an integer x being greater than zero or less than or equal to zero, which is proved by the rule, **law**. The proof is valid since what the rule proves is a positive formula. The second example is Fermat's last theorem, that is, for all integers i>=3, there exist positive numbers x, y, and z, such

that the addition of x to the power of i and y to the power of i equals to z to the power of i. We will assume that the proof will be provided later and prove it by our "wild card" rule, **arithmetic**. If the proofs are written in our grammar, they would look something like:

predicate > : nat, nat	predicate >= : nat, nat
	predicate = : nat, nat
>(x,0) or - >(x,0) by law	predicate + : nat, nat
	predicate ^ : nat, nat
	>=(i, 3) ->
	=(+(^(x,i), ^(y,i)), ^(z,i)) by arithmetic

where x, y, z, and i are some natural numbers.

3.1.2 Output Proof

The output of the proof checker is a pair consisting of a list of predicates used in a proof and a checked proof tree. Figure 3.1.2 shows *Standard ML* types and datatypes that define a proof. The names of types and datatypes are designed to be self-explanatory and to be matched to the grammar shown in Figure 3.1.1. A proof can be represented in one of two forms, either with a **pf** constructor or with a **block** constructor. If a proof has assumptions, that is, the proof of $\Gamma \vdash \phi$, then it is with a **block** constructor listing the assumptions along with the proof; otherwise, it is the proof of $\vdash \phi$, and it consists of a **pf** constructor with an inference rule used for the proof and the sequent of the proof.

The sequent, $\Gamma \vdash \phi$, is defined as a pair consisting of context Γ and proposition ϕ, where the context is the list of binders, each of which is an association of an identifier and a proposition, and the proposition is the conclusion.

```
type word = string ;
type identifier = word ;
type binder  = identifier * prop ;
type context = binder list ;
type sequent = context * prop ;
datatype prop = andd of prop * prop
              | or of prop * prop
              | hook of prop * prop
              | nott of prop
              | univ of identifier * prop * prop
              | exist of identifier * prop * prop
              | predicate of word * (expr list)
              | listof of prop
              | nat_type
              | wild_card_type
and       expr = nil
              | cons of expr * expr
              | num of word
              | id of identifier
              | succ of expr ;

datatype proof = pf of rule * sequent |  block of (binder list) * proof
and       rule  = assume of identifier
              | andI of proof * proof
              | andE of binder * binder * proof * proof
              | orIL of proof
              | orIR of proof
              | orE of proof * binder * proof * binder * proof
              | hookI of binder * proof
              | hookE of proof * proof
              | univI of binder * proof
              | univE of proof * expr
              | existI of expr * proof
              | existE of binder * binder * proof * proof
              | listE of proof * proof * binder * binder * binder * proof
              | mk_expr of expr
              | law
              | arithmetic ;
```

Figure 3.1.2 *Standard ML* datatype proof

The datatype rule corresponds to each rule defined in the grammar except that a new rule is introduced. The rule **mk_expr**, which reads as "make expression," is used to justify proofs for expressions rather than propositions by checking the types of the expressions.

Figure 3.2 shows a block diagram of the proof checker. It is divided into three modules, and each module is briefly described below.

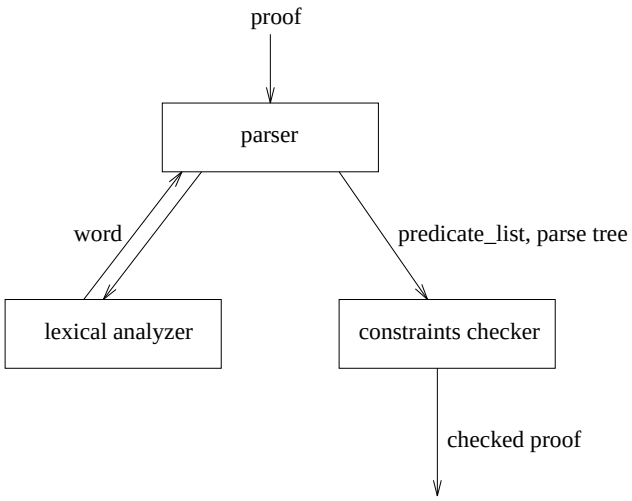


Figure 3.2 Block Diagram of the Proof Checker

3.1.3 Lexical Analysis Module

The lexical analyzer is in the file lex.sml. This program maps a sequence of ASCII characters read from an input file into words. A word is a string of characters, and words are separated by one or more blanks (or returns). The main function **lexical_analyze** reads enough characters from the input file to build one word.

3.1.4 Syntax Analysis Module

The syntax analyzer (parser) is in the file parse.sml and provides the function **parse**. This program maps a sequence of words into a parse tree. A call to the lexical

analyzer functions are made whenever a word is needed. The main function **parse** takes an input file with a proof, then parses the proof to build a parse tree. The output is a pair consisting of the list of the predicates used in the proof and the parse tree.

Example:

The input to the parser is an input file which contains a proof written in the proof language syntax. We will again use the same example proof of P⊃Q, P ⊢ P∧Q. Figure 3.1.1a shows the proof written in the grammar, and we will use it as an input to demonstrate the parser. The output of the parser is a pair consisting of the list of predicates P and Q and the parse tree of the *Standard ML* datatype **parse_tree** shown below:

```
([("P",[]),("Q",[])],
 proves (using [],
         blockp
             ([("a",hook (predicate ("P",[]),predicate ("Q",[]))),
               ("b",predicate ("P",[]))],
             proves (using [],
                     basicp (andd (predicate ("P",[]),predicate ("Q",[])),
                             andIp
                             (proves (using [],embedp "b"),
                              proves (using [],
                                      basicp(predicate ("Q",[]),
                                             hookEp
                                             (proves (using [],embedp "a"),
                                              proves (using [],embedp "b")))))))))
```

The first two lines of the input are changed to the list of two pairs for predicates P and Q as shown in the first line of the above output. The parse tree is defined by a **proves** constructor, which contains the list of the proof abstractions with a **using** constructor and the body of the proof. In the above example, the body of the proof is either with a **blockp** constructor or a **basicp** constructor. If the proof has local assumptions, it is with a **block** constructor indicating the scope of the assumptions; otherwise, a **basicp** shows what proposition is proved by which inference rule. Appendix A provides the source files for the complete definition of the datatype.

3.1.5 Semantic Analysis Module

The constraints checker is in the file constraints.sml and provides the function **check**. This program enforces the constraints defined by predicate logic inference rules on the parse tree. The main function **check** takes a pair as input; the first component defines the primitive predicates used in the proof, and the second is the parse tree of the proof, that is, the output pair of the function **parse** is passed to the function **check**. Then, the semantics of the parse tree are type-checked according to the inference rules. We will use the inference rules described in the previous chapter as type-checking rules; in this way, the enforcement of the constraints can be mechanically examined.

If it successfully passes all the checks, the list of the predicates paired with the correct proof tree is produced as an output; otherwise an error message is given to the user. The output proof tree is defined as the *Standard ML* datatype **proof** in Figure 3.1.2.

Example:

Using the output pair of the example proof in the previous section as input, the function check produces the pair consisting of the predicate list and the proof tree as follows:

```
([("P",[]),("Q",[])],
block ([("a",hook (predicate ("P",[]),predicate ("Q",[]))), ("b",predicate ("P",[]))],
pf( andI( pf( assume "b",
                ([("b",predicate ("P",[])), ("a",hook (predicate ("P",[]),predicate ("Q",[])))],
                 predicate ("P",[]))),
            pf( hookE( pf( assume "a",
                                ([("b",predicate ("P",[])),
                                  ("a",hook (predicate ("P",[]),predicate ("Q",[])))],
                                 hook (predicate ("P",[]),predicate ("Q",[])))),
                           pf( assume "b",
                                ([("b",predicate ("P",[])),
                                  ("a",hook (predicate ("P",[]),predicate ("Q",[])))],
                                 predicate ("P",[])))),
                ([("b",predicate ("P",[])),
                  ("a",hook (predicate ("P",[]),predicate ("Q",[])))],
                 predicate ("Q",[])))),
    ([("b",predicate ("P",[])),
      ("a",hook (predicate ("P",[]),predicate ("Q",[])))],
     andd (predicate ("P",[]),predicate ("Q",[])))))
```

The significant difference between the intermediate parse tree produced by the parser and the output proof tree of the constraints checker is that the latter contains a sequent rather than a predicate at each node.

The next two sections describe the main modules of the system, the realizer and the translator, which implement the realizability theory to convert a proof to programming code by using Hatcliff's algorithm. A few changes are made to the algorithm in Figure 2.5.2, and they are listed as follows:

- Lambda abstraction

  A proof by the implication introduction or universal introduction rule is translated into a lambda abstraction of the form: $\lambda q \in [\![\phi]\!].[\![p]\!]$ or $\lambda x \in [\![\tau]\!].[\![p]\!]$ respectively. In our translation process, the resulting programming code is completely separated from its specification, and it does not contain any typing information. Therefore we will discard the typing information $[\![\phi]\!]$ and $[\![\tau]\!]$ from the translated programming code. The lambda abstraction we will then use in our system is $\lambda q.[\![p]\!]$ and $\lambda x.[\![p]\!]$.

- List elimination rule

  Instead of mathematical induction, that is, elimination rule for natural numbers, we will introduce list induction, which is the list elimination rule. It is similar to the elimination rule, and it is defined as follows [16]:

  $$[\![\text{ListE l P0+ (a, b, c+) p1+}]\!] \Rightarrow \text{listrec } [\![l]\!] \; [\![p0^+]\!] \; (a, b, c) \; [\![p1^+]\!]$$

  where l is a proof for a list, p0 is a proof of the initial case for l being nil, and p1, together with $a \in$ nat, $b \in$ listof nat, and the induction hypothesis, c, is a proof of the induction case for l being cons a b. Note that the propositions that P0 and P1 are proving and the proposition of the identifier c are all the same, thus the polarities of P0, c, and P1 are the same. If they are negative, the translation results in an empty program.

3.2 Module Realizer

The realizer is in the file realize.sml and provides the functions **calculate_polarity** and **attach_polarity**. Once a correct proof tree is built by the proof checker, the realizer takes it as input, calculates the polarity of the proof using the realizability theory, and then attaches it to the proof. This is done recursively to all the subproofs of the proof. The output is the proof tree with polarity, intermediate data which is used for the translation to a programming code in the translator.

3.2.1 Polarities of Propositions and Proofs

As described in Chapter 2, a polarity is either positive (+) or negative (−) showing whether a proposition we are proving contains computational information, which is considered important, or not. Polarity is defined as the *Standard ML* datatype **polarity** shown below.

```
datatype polarity = positive | negative ;
```

Recall from Chapter 2 that a proof is said to be positive if it proves a positive proposition; otherwise it is said to be negative. Therefore, in order to know a polarity of a proof, all we need to do is to find out the polarity of its type, that is, the proposition it derives. The mechanism to calculate a polarity of a formula described in Section 2.5.2 is implemented recursively as the function **calculate_polarity**:

```
fun calculate_polarity (phi) =
   case phi of
   andd(phi1, phi2) => let val v1 = calculate_polarity (phi1) in
                          let val v2 = calculate_polarity (phi2)
                          in  polarity_or (v1, v2) end end
|  or(_, _) => positive
|  hook(_, phi2) => calculate_polarity (phi2)
|  nott(_) => negative
|  univ(_, _, phi1) => calculate_polarity (phi1)
|  exist(_, _, _) => positive
|  predicate(_, _) => negative
|  listof(_) => negative
|  nat_type => negative
|  _ => raise error_realizer_calculate_polarity ;
```

The function **calculate_polarity** calls the function **polarity_or** which works in the same way as logical or operator except that truth values are polarities, that is, true and false correspond to positive and negative respectively.

3.2.2 Proof with Polarity

Once the polarity of the proposition is calculated, it could just be attached to the corresponding proof producing a proof tree with its polarity; however, the purpose of producing this intermediate data, **proof_p**, is to make the translation process simpler and easier. As stated in Hatcliff's algorithm in Figure 2.5.2, although a proof contains some positive parts, in some cases it is translated into an empty program represented as an asterisk, *. If we know that a proof will be an empty program, then we can just ignore translation, possibly speeding up the process.

How do we know which proof will be translated into an empty program? According to Hatcliff's algorithm, each inference rule, except the disjunction introduction rule and the existential introduction rule, has one or two subproofs whose polarities decide whether the result of the translation is an empty or a non-empty program. The disjunction introduction rule and the existential introduction rule are considered special cases since they always prove positive propositions, and the translations never result in empty programs. Note that there are some possible cases which are not indicated in the algorithm. All these cases fall into the case of ⟦p⁻⟧ resulting in an empty program. Let's observe each inference rule in the algorithm more carefully.

Conjunction Introduction: ⟦∧-I p1 p2⟧

If both P1 and P2 are negative, the translation results in an empty program.

Conjunction Elimination: ⟦∧-E (q1, q2)=p1 in p2⟧

If p2 is negative, the translation results in an empty program.

Disjunction Introduction Left: ⟦∨-I1 p⟧

This is one of the three special cases where its translation never results in an empty program since the proof is always positive. The translation can be recognized as two possible cases:

1. ⟦∨-I1 p^+⟧ ⇒ inl(⟦p⟧)
2. ⟦∨-I1 p^-⟧ ⇒ inl(*)

Note in case 2 that the "tag" *inl* is necessary.

Disjunction Introduction Right: ⟦∨-I2 p⟧

This is another special case. Similar to the above disjunction introduction left rule, the translation can be recognized as two possible cases:

1. ⟦∨-I2 p^+⟧ ⇒ inr(⟦p⟧)
2. ⟦∨-I2 p^-⟧ ⇒ inr(*)

Note in case 2 that the "tag" *inr* is necessary.

Disjunction Elimination: ⟦∨-E p of q1.p1 ‖ q2.p2⟧

If both p1 and p2 are negative, the translation results in an empty program. Note that p1 and p2 prove the same proposition.

Implication Introduction: ⟦⊃-I q∈ϕ.p⟧

If p is negative, the translation results in an empty program.

Implication Elimination: ⟦⊃-E p1 p2⟧

If p1 is negative, the translation results in an empty program. Note that p1 is a proof of a proposition of the form ϕ1⊃ϕ2. If ϕ2 is negative, p1 is also negative.

Universal Introduction: ⟦∀I x∈τ.p⟧

If p is negative, the translation results in an empty program.

Universal Elimination: ⟦∀E p t⟧

If p is negative, the translation results in an empty program.

Existential Introduction: ⟦∃I t p⟧

This is one of the three special cases where its translation never results in an empty program since the proof is always positive. There are two cases:

1. $⟦∃I\ t\ p^{+}⟧ \Rightarrow (⟦t⟧, ⟦p^{+}⟧)$
2. $⟦∃I\ t\ p^{-}⟧ \Rightarrow ⟦t⟧$

Existential Elimination: ⟦∃E (x, q)=p1 in p2⟧

If p2 is negative, the translation results in an empty program.

List Elimination: ⟦ListE l p0 (a, b, c) p1⟧

If p0, c, and p1 are negative, the translation results in an empty program. Note that the propositions that p0 and p1 are proving and the proposition of the identifier c are all the same.

From the above observation, we can attach the polarity based on whether the resulting programming code is empty or non-empty except for the three special cases, where we will attach the polarity of the subproof (there is only one subproof for these particular cases) for its immediate translation. This mechanism of calculating and attaching polarities based on this observation is done by the function **attach_polarity**. The actual implementation has the following general pattern-matching format for each inference rule used in the proof, and all subproofs of the proof are recursively processed.

```
fun attach_polarity (
        ...
|   attach_polarity ( pf( orIL(p), (Gamma, or(phi1, phi2)) )) =
        let val p' = attach_polarity (p) in
        let val v  = calculate_polarity (phi1)
        in  pf_p( orIL_p(p'), (Gamma, or(phi1, phi2)), v )
        end end

|   attach_polarity ( pf(
        ...
```

The output of this function is called the *Standard ML* datatype **proof_p**, which reads as "proof with polarity," and it is defined in the next section. Once a polarity is attached, the translation of the proof is straight forward by using the algorithm.

3.2.3 Input/Output Specifications

The input to the realizer is a proof, which is the output from the proof checker, defined in Figure 3.1.3. The output of the realizer is a proof tree with its polarity, which is the same as the input proof except that now the polarity is embedded in the proof. It is defined as the *Standard ML* datatype **proof_p** shown in Figure 3.2.3. Since polarities of each subproof are calculated recursively, they are embedded only at the top level, making the datatype simple and reducing the data storage space.

```
datatype proof_p = pf_p of rule_p * sequent * polarity
                 | block_p of (binder list) * proof_p

and      rule_p  = assume_p of identifier
                 | andI_p of proof_p * proof_p
                 | andE_p of binder * binder * proof_p * proof_p
                 | orIL_p of proof_p
                 | orIR_p of proof_p
                 | orE_p  of proof_p * binder * proof_p * binder * proof_p
                 | hookI_p of binder * proof_p
                 | hookE_p of proof_p * proof_p
                 | univI_p of binder * proof_p
                 | univE_p of proof_p * proof_p
                 | existI_p of proof_p * proof_p
                 | existE_p of binder * binder * proof_p * proof_p
                 | listE_p of proof_p * proof_p * binder * binder * binder * proof_p
                 | mk_expr_p of expr
                 | law_p
                 | arithmetic_p ;
```

Figure 3.2.3 *Standard ML* datatype proof_p

3.3 Module Translator

The translator is in the file translate.sml and provides the function **translate**, which is the actual implementation of Hatcliff's algorithm for the realizability theory to translate a proof to programming code. In addition to a polarity embedded with a proof by the function **attach_polarity**, a call to the function **calculate_polarity** is made as needed during the process of the translation. Both functions are imported from the realizer described in the previous section. The translation is done recursively for all subproofs of a proof depending on each inference rule used in the proof. The general format of the function translate is given below, where **r** and **s** indicate the rule and sequent respectively:

```
fun translate (
        ...
|   translate ( pf_p( r1, s, negative ) ) = (empty, s)
|   translate ( pf_p( r1, s, positive ) ) =
        ...
|   translate ( pf_p( r2, s, negative ) ) = (empty, s)
|   translate ( pf_p( r2, s, positive ) ) =
        ...
```

3.3.1 Translation of Proofs

The implementation of the translation of each inference rule in Hatcliff's algorithm is described below. The inference rules and resulting programming codes are represented in terms of the datatypes used in the system, which correspond to the original algorithm shown in Figure 2.5.2.

Conjunction Introduction: ⟦∧-I p1 p2⟧

There are four possible cases to be considered:

1. ⟦andI $p1^+$ $p2^+$⟧ ⇒ pair(⟦$p1^+$⟧, ⟦$p2^+$⟧)
2. ⟦andI $p1^+$ $p2^-$⟧ ⇒ ⟦$p1^+$⟧
3. ⟦andI $p1^-$ $p2^+$⟧ ⇒ ⟦$p2^+$⟧
4. ⟦andI $p1^-$ $p2^-$⟧ ⇒ *

For the first three cases, the attached polarity is positive. Thus, a call to the function **calculate_polarity** is made for p1 and p2. Then, depending on their polarities, p1 and/or p2 are translated further. If both p1 and p2 are positive, the translation results in a pair consisting of two translated programs as in case 1. If the attached polarity is negative as in case 4, an empty program is returned.

Conjunction Elimination: ⟦∧-E (q1, q2)=p1 in p2⟧

There are five possible cases to be considered:

1. ⟦andE ($q1^+$, $q2^+$)=p1 in $p2^+$⟧ ⇒ let2 (q1, q2)=⟦p1⟧ in ⟦$p2^+$⟧
2. ⟦andE ($q1^+$, $q2^-$)=p1 in $p2^+$⟧ ⇒ let1 q1=⟦p1⟧ in ⟦$p2^+$⟧
3. ⟦andE ($q1^-$, $q2^+$)=p1 in $p2^+$⟧ ⇒ let1 q2=⟦p1⟧ in ⟦$p2^+$⟧
4. ⟦andE ($q1^-$, $q2^-$)=p1 in $p2^+$⟧ ⇒ ⟦$p2^+$⟧
5. ⟦andE (q1 , q2)=p1 in $p2^-$⟧ ⇒ *

In the first four cases, only the positive components of p1 are substituted in p2. The **let2** constructor used in case 1 substitutes two variables, and the **let1** constructor used in cases 2 and 3 substitutes one variable. In case 4, no substitution occurs since both components of p1 are negative. Finally, if the attached polarity is negative as in case 5, an empty program is returned.

Disjunction Introduction Left: ⟦∨-I1 p⟧

There are two possible cases to be considered:

1. ⟦orIL p^+⟧ ⇒ inl(⟦p^+⟧)
2. ⟦orIL p^-⟧ ⇒ inl(*)

The disjunction introduction rules are used to prove the proposition of the form $\phi 1 \vee \phi 2$, and the proof is always positive. If the attached polarity, that is, the polarity of p, is positive as in case 1, p is recursively translated and is "tagged" with an **inl** constructor. Otherwise, an empty program is "tagged" as in case 2.

Disjunction Introduction Right: ⟦∨-I2 p⟧

There are two possible cases to be considered:

1. ⟦orIR p^+⟧ ⇒ inr(⟦p^+⟧)
2. ⟦orIR p^-⟧ ⇒ inr(*)

The translation is done in the same way as the disjunction introduction left rule described above.

Disjunction Elimination: ⟦v-E p of q1.p1 ‖ q2.p2⟧

There are two possible cases to be considered:

1. ⟦orE p of q1.$p1^+$ ‖ q2.$p2^+$⟧ ⇒ casee ⟦p⟧ of (inl q1).⟦$p1^+$⟧ ‖ (inr q2).⟦$p2^+$⟧
2. ⟦orE p of q1.$p1^-$ ‖ q2.$p2^-$⟧ ⇒ *

The attached polarity indicates the polarity of p1, which is the same as the polarity of p2. If it is positive, the proof is translated into a case statement, where the first case is q1 and the second case is q2 as in case 1. Here, p is a proof of a proposition of the form $\phi 1 \vee \phi 2$, where each component is called q1 of type $\phi 1$ and q2 of type $\phi 2$. The three subproofs, p, p1, and p2, are recursively translated. If the attached polarity is negative, both p1 and p2 are translated into empty programs causing both cases to become identical. Therefore, the translation results in an empty program as in case 2.

Implication Introduction: ⟦⊃-I q∈ϕ.p⟧

There are three possible cases to be considered:

1. ⟦hookI q∈ϕ⁺.p⁺⟧ ⇒ lambda q.⟦p⁺⟧
2. ⟦hookI q∈ϕ⁻.p⁺⟧ ⇒ ⟦p⁺⟧
3. ⟦hookI q∈ϕ .p⁻⟧ ⇒ *

The attached polarity indicates the polarity of the subproof p. If it is positive as in cases 1 and 2, p is recursively translated, and a call to the function **calculate_polarity** is made for ϕ. If ϕ is positive, the translation results in a lambda abstraction as in case 1; otherwise, it is solely the translation of p as in case 2. If the attached polarity is negative, an empty program is returned as in case 2.

Implication Elimination: ⟦⊃-E p1 p2⟧

There are three possible cases to be considered:

1. ⟦hookE p1⁺⊃⁺ p2⟧ ⇒ apply ⟦p1⁺⟧ • ⟦p2⁺⟧
2. ⟦hookE p1⁻⊃⁺ p2⟧ ⇒ ⟦p⁺⟧
3. ⟦hookE p1 ⊃⁻ p2⟧ ⇒ *

The implication elimination corresponds to an application. Here, p proves a proposition of the form ϕ1⊃ϕ2, and the attached polarity indicates the polarity of the proof, that is, the polarity of ϕ2. If it is positive as in cases 1 and 2, the translation depends on the polarity of ϕ1, that is, the polarity of p2. A call to the function **calculate_polarity** is made for ϕ1, and if ϕ1 is positive, p1 and p2 are recursively translated, and an application of the translated p1 to the translated p2 results as in case 1; otherwise it is solely the translation of p1 as in case 2. If the attached polarity is negative, an empty program is returned as in case 3.

Universal Introduction: ⟦∀I x∈τ.p⟧

There are two possible cases to be considered:

1. ⟦univI x∈τ.p⁺⟧ ⇒ lambda x.⟦p⁺⟧
2. ⟦univI x∈τ.p⁻⟧ ⇒ *

Similar to the implication introduction, the universal introduction corresponds to a lambda abstraction except that the variable x is a data type rather than a proposition.

Therefore, x will not appear in the translation since Hatcliff specifically distinguishes data types from propositions, and data types are not involved in the translation. The attached polarity indicates the polarity of the subproof p. If it is positive, the translation results in lambda abstraction, and p is recursively translated as in case 1; otherwise an empty program is returned as in case 2.

Universal Elimination: ⟦∀E p t⟧

There are two possible cases to be considered:

1. ⟦univE p^+ t⟧ ⇒ apply ⟦p^+⟧•⟦t⟧
2. ⟦univE p^- t⟧ ⇒ *

Similar to the implication elimination, the universal elimination corresponds to application. The attached polarity is the polarity of the subproof p. If it is positive, p and t are recursively translated, and the translation of the proof is an application of the translated p to the translated t as in case 1. Note that t is an expression, which is either an identifier, number, or list. Although an expression is in fact a term, we differentiate it in the translation for its clarity. Thus, the translation of t is tagged with an **exprt** constructor:

⟦t⟧ ⇒ exprt(t)

If the polarity is negative, an empty program is returned as in case 2.

Existential Introduction: ⟦∃I t p⟧

There are two possible cases to be considered:

1. ⟦existI t p^+⟧ ⇒ pair(⟦t⟧, ⟦p^+⟧)
2. ⟦existI t p^-⟧ ⇒ ⟦t⟧

Similar to the conjunction introduction, the existential introduction corresponds to a pair consisting of an instance and its proof. The existential introduction is used to prove the proposition of the form $\exists x \in \tau . \phi$, and the proof is always positive. Thus, the attached polarity is the polarity of the subproof p. If it is positive, p is recursively translated and paired with the translated expression τ as in case 1. Otherwise, the translation of the proof is solely the translation of expression t as in case 2.

Existential Elimination: ⟦∃E (x, q)=p1 in p2⟧

There are three possible cases to be considered:

1. ⟦existE (x, q+)=p1+ in p2+⟧ ⇒ let2 (x, q)=⟦p1+⟧ in ⟦p2+⟧
2. ⟦existE (x, q−)=p1− in p2+⟧ ⇒ let1 x=⟦p1⟧ in ⟦p2+⟧
3. ⟦existE (x, q)=p1 in p2−⟧ ⇒ *

Similar to the conjunction elimination, the existential elimination corresponds to substitution. Here, p1 proves a proposition of the form $\exists x \in \tau.\phi$, and it is always positive. The attached polarity is the polarity of the proof itself, that is, the polarity of the subproof p2. If it is positive, p2 is recursively translated, and a call to the function **calculate_polarity** for variable q is made. Depending on the polarity of q, the translation of the proof is either the substitution of one or two variable(s). If q is positive, the subproof p1 is recursively translated, and the translation is the substitution of both x and q for p1 in p2 as in case 1; otherwise it is the substitution of x for p1 in p2 as in case 2. If p2 is negative, an empty program is returned as in case 3.

List Elimination: ⟦ListE l p0 (a, b, c) p1⟧

There are two possible cases to be considered:

1. ⟦ListE l p0+ (a, b, c+) p1+⟧ ⇒ listrec ⟦l⟧ ⟦p0+⟧ (a, b, c) ⟦p1+⟧
2. ⟦ListE l p0− (a, b, c−) p1−⟧ ⇒ *

The polarity of p0, c, and p1 are the same, where the subproof p0 is a proof for the initial case ϕ(nil), c is an induction hypothesis ϕ(b), and the subproof p1 is a proof for the induction case ϕ(cons a b), and the attached polarity is the polarity of the proof for ϕ(l). If it is positive, list l, p0, and p1 are translated further to produce a list recursion as in case 1; otherwise an empty program is returned as in case 2.

The above mechanism is implemented as the function **translate**, and the output is a translated programming code, which is called a term, paired with a sequent of the original proof. The sequence of the proof is not changed in the translation process, and the translated programming code carries it as its "type tag," that is, the specification of the program.

3.3.2 Input/Output Specifications

The input to the translator is the *Standard ML* datatype **proof_p** described in Figure 3.2.3, which is the output of the function **attach_polarity**. The output of the translator is a pair consisting of a programming code (term) and a sequent. The programming codes translated from proofs are defined as the *Standard ML* datatype **term** shown in Figure 3.3.2.

```
type number = string ;
datatype term = empty
              | ident of identifier
              | pair of term * term
              | inl of term
              | inr of term
              | let1 of identifier * term * term
              | let2 of identifier * identifier * term * term
              | casee of term * identifier * term * identifier * term
              | lambda of identifier * term
              | apply of term * term
              | listrec of term * term * identifier * identifier * identifier * term
              | numt of number
              | lawt of prop
              | exprt of expr ;
```

Figure 3.3.2 *Standard ML* datatype term

A **let1** constructor substitutes one identifier whereas a **let2** constructor substitutes two identifiers. If a positive axiom or theorem is used in a proof, it is translated into a term with a **lawt** constructor with the specification of the axiom or the theorem.

3.4 Module Linker

The linker is in the file link.sml and provides the function **link**. The linker links two terms into one by using the cut-rule and β-rule (substitution) described in Chapter 2. Terms are the output programming code from the translator described in the previous section. The idea of the linker is that the linking process can be done by checking only the specifications of the programs, but not the programs themselves. This implementation is possible by applying the cut-rule. Figure 3.4 shows the block diagram of the linker with its input and output, and the linking process is described in the following sections.

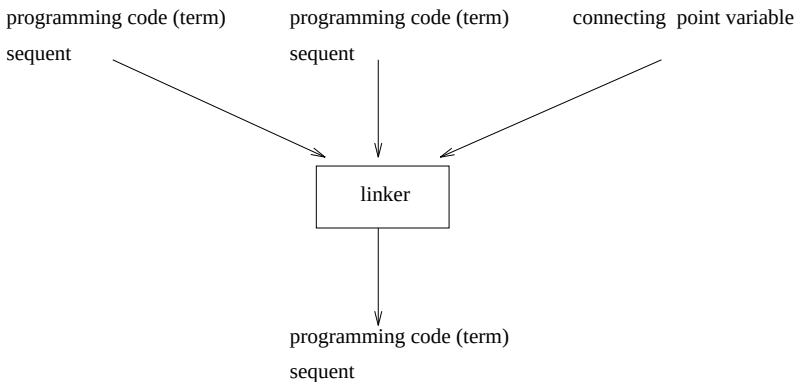


Figure 3.4 Block Diagram of the Linker

3.4.1 Cut Rule

The cut-rule states that if there are two proofs for $\phi 1$ and $\phi 2$, where a variable x of type $\phi 1$ can be used to prove P2, then we can combine these two proofs into one which proves P2 from Γ and Δ assumption lists, where all the occurrences of x in P2 are substituted with P1. To get the result $\phi 2$ we use β-rule. Each programming code produced from the translator is paired with its original type, that is, the sequent of the

proof from which the programming code is translated. The cut-rule forces some constraints on the sequent, thus we can use it as a typing rule as shown in Figure 3.4.1. Then, our implementation of the linker is based on the rule, and actually it is an implementation of the rule itself.

$$\frac{\Gamma \vdash P1 : \phi 1 \qquad \Delta,\ x : \phi 1 \vdash P2 : \phi 2}{\Gamma,\ \Delta \vdash [P1/x]P2 : \phi 2}$$

Figure 3.4.1 Cut Rule as Typing Rule

3.4.2 Three Conditions of Cut Rule

According to the cut-rule, we need two programs (terms), t1 and t2, each with its sequent, s1 and s2 respectively, and a variable x which indicates where the programs will be connected. Here, s1 is $\Gamma \vdash \phi 1$ and s2 is Δ, $x{:}\phi 1 \vdash \phi 2$ as shown in Figure 3.4.1. In order to apply the cut-rule to the linker, there are three conditions that must be met:

1. a connecting point variable x is in the context of s2,
2. the type of x matches the type of the conclusion of s1, and
3. no variable has the same name, but has different types in the merged context.

There are several auxiliary functions provided to check the above conditions. The main function **link** takes two terms, t1 and t2, two sequents, s1 and s2, and a variable x as inputs, and produces a linked term paired with its sequent as output by calling two functions **cut** and **beta**. The function **cut** takes the two input sequents, s1 and s2, and the variable x to check the three conditions and produces a new sequent as an output. The first condition is checked by a call to the function **find_x**, which looks for x in s2. If x is in s2, it returns the type of x, that is, the proposition bound to x; otherwise an error message is generated. Depending on the output of the function **find_x**, the second condition is checked in the function **cut**. Finally, the third condition is enforced by the function **dup_x** as the two contexts Γ and Δ of s1 and s2 respectively are merged.

The third condition has the most tasks of the three. First, the function **delta** is called to extract the assumption x from Δ, then a call to the function **gamma-delta** is made to merge Γ and Δ without the binder of x. The convention of the ordering in a context is that it is built from the most local (newest) assumption to the most global (oldest) one. Here, the ordering is important. Since a context is implemented as a list, it consists of the most local one at the head and the most global one at the tail. As the cut-rule indicates, the proof of the assumption x is derived from Γ, Δ being considered newer than Γ. Therefore, in order to merge Γ and Δ properly, each assumption in Δ is extracted from the head of Δ, the rest of the assumptions in Δ are recursively and correctly merged to Γ, and then the head element of Δ is put into the head of the merged list, Γ and Δ.

Also, the merged context should have no duplicate assumptions. If Γ and Δ has the same assumptions, then the ones in Δ are deleted since they are more local than the ones in Γ. As the function **gamma-delta** takes the head element of Δ, the duplication check is done by a call to the function **dup_x**. The boolean function **dup_x** takes an assumption from Δ and Γ as inputs. It recursively checks whether the variable of the assumption is already in Γ. If it is in Γ and the type of the variable, that is, the proposition associated with the variable is also matched, then it returns true. If the variables have the same name, but their propositions do not match, then an error exception is raised. If the variable is not in Γ, then false is returned, and a call to the function **add_x** is made to put the assumption in Γ. This process can be done recursively as follows:

```
fun gamma-delta (Gamma, []) = Gamma
|   gamma-delta (Gamma, b::tl) =
        let val dup = dup_x (b, Gamma) in
        let val Gamma' = gamma_delta (Gamma, tl)
        in  if dup then Gamma'
                   else b::Gamma'
        end end ;
```

If all the three conditions are satisfied, the two terms can be linked by calling the function **beta**. It uses the two input terms, t1 and t2, the connecting point variable x, and the new sequent s produced by the function **cut** and links the terms by introducing a new constructor **lett** to the *Standard ML* datatype **term**. A **lett** constructor is defined as:

```
lett of identifier * term * term
```

The new term **lett** is the implementation of β-rule. The function **beta** is shown below:

```
fun beta (t1, t2, x, s) = ( lett(x, t1, t2), s )
```

Like the other let constructors in the datatype **term**, a **lett** is read as "let x=t1 in t2," which can be represented as the substitution notation [t1/x]t2. We do not actually substitute all the occurrences of x in t2 with t1, thus x's in t2 stay as they are and a **lett** constructor states that x is to be substituted with t1 in t2. As stated in the rule, the substitution is done one variable at a time, hence the linking process also needs to be done one variable at a time, in which case the resulting linked term will have several nested **lett** constructors.

3.4.3 Input/Output Specifications

The inputs to the linker are two terms each with its sequent, and a connecting point variable, where terms are the datatype **term** shown in Figure 3.3.2 together with a new constructor **lett** introduced in the previous section. The output is a pair consisting of the linked term represented by a **lett** constructor and a new sequent.

Chapter 4
Case Study - Insertion Sort

In the previous chapter, we have shown how our system translates a logical proof into a programming code by using a small input proof as an example. In this chapter, we will expand the input proofs to more practical and complicated cases. The main purpose of the conversion of formal logic proofs to programming codes is to write correct programs. Therefore, we should be able to apply the conversion process to an example such as a sorting program for lists, that is, the proof of a sorting specification. Among different searching techniques such as selection, insertion, merge, and quick sorts, we will choose an insertion sort. We will use the approaches described in [16]. To simplify the matter, only the list of natural numbers, which is represented as **listof nat**, is considered.

4.1 Properties of Lists

We will use the two properties of lists, the predicates *perm* and *sorted*, which are similar to the predicates *perm* and *ord* in [16] with a few modifications.
For l, m∈listof nat, the predicate perm(l,m) states that l and m are permutations of one another, and the predicate sort(l) states that l is a sorted list. Figure 4.1 describes a set of ten assumptions, from z1 to z10, about perm and sorted.

The first three assumptions, z1, z2, and z3, state the reflexivity, symmetry, and transitivity of *perm* respectively. The assumption z4 states that after adding the same element to two lists, which are permutations of one another, the lists are still permutations of one another. The assumption z5 states that changing the order of the first and second elements of a list does not change its content. The assumptions z6 and z7 state that nil list and a single element list are sorted. The assumption z8 states that if a non-empty list is sorted, its tail is also sorted. The assumption z9 states that if there is a non-empty sorted list and a number which is less than the smallest number in the list, the list with the number added to its head is sorted. The assumption z10

states that when number a is added to the head of list m, the resulting list cons a m is sorted for the following reasons:

- since list cons a l is sorted, number a is the smallest in the list l,
- number a is smaller than or equal to number b,
- list m consists of number b and all the elements of list l, and
- number a is the smallest in list m.

z1 : ∀l∈listof nat. perm(l, l)
z2 : ∀l∈listof nat. ∀m∈listof nat. (perm(l, m) ⊃ perm(m, l))
z3 : ∀l∈listof nat. ∀m∈listof nat. ∀n∈listof nat.
    (perm(l,m) ∧ perm(m,n)) ⊃ perm(l,n)
z4 : ∀a∈nat. ∀l∈listof nat. ∀m∈listof nat. perm(l,m) ⊃ perm(cons a l, cons a m)
z5 : ∀a∈nat. ∀b∈nat. ∀l∈listof nat. perm(cons a (cons b l), cons b (cons a l))
z6 : sorted(nil)
z7 : ∀a∈nat. sorted(cons a nil)
z8 : ∀a∈nat. ∀l∈listof nat. sorted(cons a l) ⊃ sorted(l)
z9 : ∀a∈nat. ∀b∈nat. ∀l∈listof nat.
    (sorted(cons a l) ∧ <(b, a)) ⊃ sorted(cons b (cons a l))
z10: ∀a∈nat. ∀b∈nat. ∀l∈listof nat. ∀m∈listof nat.
    (sorted(cons a l) ∧ ¬ <(b, a) ∧ perm(cons b l, m) ∧ sorted(m))
    ⊃ sorted(cons a m)

Figure 4.1 A Set of Assumptions About *perm* and *sorted*

4.2 Function Insert

We can use the ten properties, z1..z10, in Figure 4.1 as assumptions to prove an insert function specification:

z1..z10 ⊢ insert ∈ ∀l∈listof nat.
    sorted(l) ⊃ (∀a∈nat. ∃m∈listof nat. sorted(m) ∧ perm(cons a l, m))

Here, *insert* is a program which inserts a number a into the appropriate place in sorted list l so that the resulting list m is still sorted. This function can be used to derive the proof of the function sort in the next section. The following section describes the main strategy of the proof.

4.2.1 Input Proof for Insert

The strategy of the proof is that we wish to prove:

sorted(l) ⊃ (∀a∈nat. ∃m∈listof nat. sorted(m) ∧ perm(cons a l, m))

for an arbitrary list l∈listof nat. The proof is by the list elimination rule, which is the list induction, and it consists of two parts: the first part is the initial case for list l being nil and the second part is the induction case for list l being cons x y for x∈nat and y∈listof nat. Each case is described further below:

1. Initial case:

This is the case for list l being nil, that is, we wish to prove:

sorted(nil) ⊃ (∀a∈nat. ∃m∈listof nat. sorted(m) ∧ perm(cons a nil))

Since we have z1..z10 ⊢ sorted(cons a nil) by using the universal elimination rule on z7 with a∈nat and z1..z10 ⊢ perm(cons a nil, cons a nil) by using the universal elimination rule on z1 with cons a nil ∈listof nat, we can utilize introduction rules for the connectives and quantifiers to get the desired result.

2. Induction case:

This is the case for list l being cons x y for x∈nat and y∈listof nat. Here, we are allowed to assume that there are a natural number x, a list of natural numbers y, and the induction hypothesis for list y:

sorted(y) ⊃ (∀a∈nat. ∃m∈listof nat. sorted(m) ∧ perm(cons a y, m)

With there assumptions, we wish to prove:

sorted(cons x y) ⊃

(∀a∈nat. ∃m∈listof nat. sorted(m) ∧ perm(cons a (cons x y), m))

After applying two introduction rules, we are to prove:

∃m∈listof nat. sorted(m) ∧ perm(cons a (cons x y), m)

The proof is by the disjunction elimination rule on the mathematical axiom:

<(a, x)∨¬<(a, x)

that is, for a, x nat, a is either less than x or greater or equal to x. Intuitively, we know that this is true by laws of mathematics. Thus, we can use the rule by **law** to prove the axiom, and then it is used in the disjunction elimination rule. There are two cases: one is the case that we assume <(a, x), and the other case is that we assume

¬<(a, x). Each case is described further.

Case 1. Assuming that a is less than x, <(a, x):

With the assumption <(a, x), we wish to prove:

∃m∈listof nat. sorted(m) ∧ perm(cons a (cons x y), m)

We have z1..z10 ⊢ sorted(cons a (cons x y)) by using the implication elimination rule involving assumptions z9, sorted(m), and <(a, x). We also have z1..z10 ⊢ perm(cons a (cons x y), cons a (cons x y)) by using the universal elimination rule on z1 with cons a (cons x y). The proof is by the existential introduction rule for list m being cons a (cons x y) followed by the conjunction introduction for the above components.

Case 2. Assuming that a is greater than or equal to x, ¬<(a, x)

With the assumption ¬<(a, x), we wish to prove the same proposition as in case 1:

∃m∈listof nat. sorted(m) ∧ perm(cons a (cons x y), m)

The proof is by the existential elimination rule on:

∃m∈listof nat. sorted(m) ∧ perm(cons a y, m),

which is derived from the induction hypothesis for list y and assumptions sorted(m) and z8. Here, we are allowed to assume:

sorted(m2)∧perm(cons a y, m2) is true for list m2∈listof nat.

With these assumptions, the existential introduction rule is used for cons x m followed by the conjunction introduction rule involving a rather long sequence of proof steps with the assumptions z3..z5 and z10.

Figure 4.2.1 shows the actual input proof for insert function written in the grammar shown in Figure 3.1.1. The properties z2 and z6 are not listed in the initial assumptions since they are not used in the proof at all. The proof is entered to the system and successfully translated into programming code, which is discussed in the next section. The natural deduction tree of the above proof for the function insert is provided in Appendix B. As described in Chapter 2, the leaves are assumptions and the root is the desired proof.

```
predicate perm : listof nat , listof nat
predicate sorted : listof nat
predicate < : nat , nat

assume
z1: ! l:listof nat. perm(l,l) ,
z3 : ! l:listof nat. ! m:listof nat. ! n:listof nat. (perm(l,m) & perm(m,n)) -> perm(l,n) ,
z4 : ! a:nat. ! l:listof nat. ! m:listof nat. perm(l,m) -> perm(cons a l,cons a m) ,
z5 : ! a:nat.! b:nat.! l:listof nat. perm(cons a (cons b l), cons b (cons a l)) ,
z7 : ! a:nat. sorted(cons a nil) ,
z8 : ! a:nat.! l:listof nat. sorted(cons a l) -> sorted(l) ,
z9 : ! a:nat.! b:nat.! l:listof nat.(sorted(cons a l) & <(b,a)) -> sorted(cons b (cons a l)),
z10: ! a:nat.! b:nat.! l:listof nat.! m:listof nat.
     (sorted(cons a l) & - <(b,a) & perm(cons b l,m) & sorted(m)) -> sorted(cons a m)
in

! l:listof nat. sorted(l) -> ( ! a:nat.? m:listof nat. sorted(m) & perm(cons a l, m) )
by !I
    ( assume l:listof nat in

        line nilcase =
        sorted(nil) -> ( ! a:nat.? m:listof nat.sorted(m) & perm(cons a nil, m) )
        by ->I ( assume p:sorted(nil) in
                  ! a:nat.? m:listof nat. sorted(m) & perm(cons a nil, m)
                  by !I ( assume a:nat in

                            ? m:listof nat.sorted(m) & perm(cons a nil, m)
                            by ?I ( cons a nil )
                                  (_ by &I (_ by !E ( embed z7 ) a )
                                           (_ by !E ( embed z1 ) cons a nil )
                                  )
                            end )
                  end )

        line conscase =

        assume x:nat, y:listof nat,
                 rec_y: sorted(y) ->
                        (! a:nat.? m:listof nat. sorted(m) & perm(cons a y, m)) in

        sorted(cons x y) ->
        ( ! a:nat.? m:listof nat. sorted(m) & perm(cons a (cons x y), m) )
        by ->I
        ( assume s:sorted(cons x y) in
```

```
! a:nat.? m:listof nat. sorted(m) & perm(cons a (cons x y), m)
by !I ( assume a:nat in

        ? m:listof nat.sorted(m) & perm(cons a (cons x y), m)
        by vE ( <(a,x) v - <(a,x) by law )

                ( assume p: <(a,x) in

                line z9' =
                ( sorted(cons x y) & <(a,x) ) -> sorted(cons a (cons x y) )
                by !E (_ by !E (_ by !E ( embed z9 ) x ) a ) y

                ? m:listof nat. sorted(m) & perm(cons a (cons x y), m)
                by ?I   cons a (cons x y)
                        (_ by &I (_ by ->E   ( use z9' )
                                             (_ by &I ( embed s ) ( embed p ))
                                 )
                                 (_ by !E ( embed z1 ) cons a (cons x y))
                        )
                 end )

                ( assume q: - <(a,x) in

                line z8' =
                sorted(cons x y) -> sorted(y)
                by !E (_ by !E ( embed z8 ) x ) y

                ? m:listof nat.sorted(m) & perm(cons a (cons x y),m)
                by ?E
                    ( ? m:listof nat.sorted(m) & perm(cons a y, m)
                    by !E (_ by ->E   ( embed rec_y )
                                      (_ by ->E ( use z8' ) ( embed s ))
                          ) a
                    )

                    ( assume   m2:listof nat,
                               bm2:sorted(m2) & perm(cons a y,m2) in

                        line z10' = ( sorted(cons x y) & - <(a,x) &
                                      perm(cons a y, m2) & sorted(m2) )
                                      -> sorted(cons x m2)
                                      by !E (_ by !E (_ by !E
                                                     (_ by !E
                                                          ( embed z10 )
```

```
                                                x
                                            ) a
                                        ) y
                                ) m2

line bm21' =   sorted(m2)
               by &E ( embed bm2 )
                     ( assume bm21: sorted(m2),
                          bm22: perm(cons a y, m2) in
                       embed bm21
                       end )

line bm22' =   perm(cons a y, m2)
               by &E ( embed bm2 )
                     ( assume bm21: sorted(m2),
                          bm22: perm(cons a y, m2) in
                       embed bm22
                       end )

line 1 = sorted(cons x m2)
         by ->E ( use z10' )
                  (_ by &I (_ by &I (_ by &I ( embed s )
                                             ( embed q )
                                    )
                                    ( use bm22' )
                           )
                           ( use bm21' )
                   )

line z3' =
( perm(cons a (cons x y),cons x (cons a y)) &
perm(cons x (cons a y), cons x m2) )
-> perm(cons a (cons x y), cons x m2)
by !E (_ by !E (_ by !E ( embed z3 ) cons a (cons x y)
                )cons x (cons a y)
       ) cons x m2

line z4' =
perm(cons x (cons a y), cons x m2)
by ->E (_ by !E (_ by !E (_ by !E ( embed z4 ) x
                                  ) cons a y
                 ) m2
        )
        ( use bm22' )
```

```
                        line z5' = perm(cons a (cons x y), cons x (cons a y))
                                by !E (_ by !E (_ by !E ( embed z5 ) a) x) y

                         line 2 = perm(cons a (cons x y), cons x m2)
                                by ->E ( use z3' ) (_ by &I ( use z5' ) ( use z4' ))

                        ? m:listof nat. sorted(m) & perm(cons a (cons x y), m)
                        by ?I cons x m2  (_ by &I ( use 1 ) ( use 2 ))
                        end )
                end )
        end )
end )
end

sorted(l) -> ( ! a:nat.? m:listof nat. sorted(m) & perm(cons a l, m) )
by listE ( embed l ) ( use nilcase ) ( use conscase )
end )
end
```

Figure 4.2.1 Input Proof for Insert

4.2.2 Output Programming Code for Insert

After passing the input proof of the function insert to the proof checker, realizer, and translator, it is converted into an insert program. Figure 4.2.2a shows the actual output term from the translator. For readability purposes, Figure 4.2.2b shows the edited version of the output by modifying some keywords and punctuations in the original output shown in Figure 4.2.2a. Figure 4.2.2c shows an insert function **insert** written in *Standard ML*. What we expect is that the output programming code for function insert matches function **insert** in Figure 4.2.2c.

Function **insert** in Figure 4.2.2c states that it takes a sorted list and an element a and inserts a into the list as follows:

> if the list is nil, then the result is the single element list [a]; or else if a is less than the head of the list, a is inserted into the head of the list; otherwise a is inserted into the proper position in the tail of the list by a recursive call to **insert** with the tail of the list and a.

This is equivalent to the programming code in Figures 4.2.2a and 4.2.2b, where we

have two inputs l and a indicated by the lambda abstraction, and the list recursion on l. If l is nil, then the result is a single element list cons a nil; else if l is cons x y and a is less than x then the result is the list cons a (cons x y); otherwise, the result is the list cons x m2, where m2 is substituted by a recursive call with the tail list y applied to a.

```
lambda ("l",
   listrec (exprt (id "l"),
            lambda ("a", exprt (cons (id "a",nill))),
            "x","y","rec_y",
                  lambda ("a",
                     casee (lawt ( or (predicate ("<",[id "a",id "x"]),
                                       nott (predicate ("<",[id "a",id "x"])))),
                              "p", exprt (cons (id "a", cons (id "x",id "y"))),
                              "q", let1 ("m2", apply (ident "rec_y", exprt (id "a")),
                                          exprt (cons (id "x",id "m2")))))))
```

Figure 4.2.2a Translated Programming Code for Insert

```
lambda l.
   listrec l of
         isnil. lambda a. cons a nil
         (iscons x y).rec_y.
                  lambda a. case (<(a, x) or not <(a,x)) of
                                    (inl p). cons a (cons x y)
                                    (inr q). let m2=rec_y a in cons x m2
```

Figure 4.2.2b Translated Programming Code for Insert (Edited)

```
fun insert [] a = [a]
|    insert x::y a = if a<x
                              then a :: x :: y
                              else let val m2=insert y a in x :: m2 end
```

Figure 4.2.2c Function Insert Written in *Standard ML*

4.3 Function Sort

We can use the insert function described in the previous section as well as the ten properties, z1..z10 in Figure 4.1, as assumptions to derive a sorting function specification:

z1..z10, insert ⊢ sort ∈ ∀l∈listof nat. ∃m∈listof nat. sorted(m) ∧ perm(m,l)

that is, **sort** is a program which sorts an arbitrary list l and produces the sorted list called m. This is a specification of an insertion sort function. The next section describes the main strategy of the proof.

4.3.1 Input Proof for Sort

The strategy of the proof is that we wish to prove:

∀l∈listof nat. ∃m∈listof nat. sorted(m) ∧ perm(m,l)

that is to show, for an arbitrary list l∈listof nat, that:

∃m∈listof nat. sorted(m) ∧ perm(m,l).

The proof is by the list elimination rule, which consists of two parts: the first part is the initial case where we are to prove: ∃m∈listof nat. sorted(m) ∧ perm(m,nil), and the second part is the induction case where we are to prove:

∃m∈listof nat. sorted(m) ∧ perm(m,cons a b).

Each case is described further below.

1. Initial case:

Since we have z1..z10 ⊢ sorted(nil,nil) by using the universal elimination rule on z1 with a nil list and z6 ⊢ z6 ∈ sorted(nil), we can utilize the conjunction introduction rule followed by the existential introduction rule to get the desired proposition:

∃m∈listof nat. sorted(m) ∧ perm(m,nil).

2. Induction case:

In this case list l is cons a b for a∈nat and b∈listof nat. Here, we are allowed to assume that there are a natural number a∈nat, a list of natural numbers b∈listof nat, and the induction hypothesis for the list b:

∃m∈listof nat. sorted(m) ∧ perm(m,b).

With these assumptions, we wish to prove:

$\exists m \in$listof nat. sorted(m) $\wedge$ perm(m,cons a b)

By using the induction hypothesis for the existential elimination rule, we are allowed to assume that sorted(m2) $\wedge$ perm(m2,cons a b) for list m2$\in$listof nat. The proof is by another existential elimination rule on:

$\exists m \in$listof nat. sorted(m) $\wedge$ perm(cons a m2, m),

which is derived from the assumption insert. We are then allowed to assume that sorted(m3) $\wedge$ perm(cons a m2, m3) for list m3$\in$listof nat. With these assumptions together with z2..z4, we can get the desired proof by using the existential introduction rule on m3 followed by the conjunction introduction rule.

Figure 4.3.1 shows the actual input proof for the function sort written in the grammar shown in Figure 3.1.1. Only the properties z1..z4 and z6, which are used in the proof, are listed in the initial assumptions. The proof is entered to the system and successfully translated into the programming code shown in the next section. Similar to the function insert, the natural deduction tree of the above proof for the function sort is provided in Appendix C. As described in Chapter 2, the leaves are assumptions and the root is the desired proof.

```
predicate perm : listof nat , listof nat
predicate sorted : listof nat

assume
insert:! l:listof nat. sorted(l) -> (! a:nat.? m:listof nat. sorted(m) & perm(cons a l, m)) ,
z1: ! l:listof nat. perm(l,l) ,
z2: ! l:listof nat.! m:listof nat. ( perm(l,m) -> perm(m,l) ) ,
z3: ! l:listof nat.! m:listof nat.! n:listof nat.  ( perm(l,m) & perm(m,n) ) -> perm(l,n) ,
z4: ! a:nat.! l:listof nat.! m:listof nat. perm(l,m) -> perm(cons a l, cons a m) ,
z6: sorted(nil)  in

! l:listof nat.? m:listof nat. sorted(m) & perm(m,l)
by !I ( assume l:listof nat in

        line nilcase =  ? m:listof nat. sorted(m) & perm(m, nil)
                        by ?I nil (_ by &I ( embed z6 ) (_ by !E ( embed z1 ) nil ))
```

```
line conscase =
assume a:nat, b:listof nat, rec_b: ? m:listof nat. sorted(m) & perm(m,b) in

? m:listof nat. sorted(m) & perm(m, cons a b)
by ?E  ( embed rec_b )
        ( assume m2:listof nat, rec_bm2:sorted(m2) & perm(m2,b) in

        line insert' =
        ? m:listof nat. sorted(m) & perm(cons a m2, m)
        by !E ( ! a:nat.? m:listof nat. sorted(m) & perm(cons a m2, m)
                by ->E (_ by !E ( embed insert ) m2 )
                        (_ by &E ( embed rec_bm2 )
                                ( assume rec_bm21:sorted(m2),
                                        rec_bm22:perm(m2,b) in
                                   embed rec_bm21
                                 end )
                        )
                ) a

        ? m: listof nat. sorted(m) & perm(m, cons a b)
        by ?E ( use insert' )
                ( assume m3:listof nat,bm3:sorted(m3) & perm(cons a m2,m3) in

                line 1 = sorted(m3)
                        by &E ( embed bm3 )
                                ( assume bm31: sorted(m3),
                                          bm32: perm(cons a m2,m3) in
                                   embed bm31
                                end )

                line z2' = perm(cons a b, m3) -> perm(m3, cons a b)
                        by !E (_ by !E ( embed z2 ) cons a b) m3

                line z4' = perm(b,m2) -> perm(cons a b, cons a m2)
                        by !E (_ by !E (_ by !E ( embed z4 ) a) b) m2

                line z2'' = perm(b,m2)
                        by ->E (_ by !E (_ by !E ( embed z2 ) m2) b)
                                (_ by &E
                                        ( embed rec_bm2 )
                                        ( assume rec_bm21: sorted(m2),
                                                  rec_bm22: perm(m2,b) in
                                             embed rec_bm22
                                          end ))
```

```
                    line z3' =
                    perm(cons a b, m3)
                    by ->E (_ by !E (_ by !E (_ by !E ( embed z3 ) cons a b
                                                    ) cons a m2
                                            ) m3
                            )
                            (_ by &I (_ by ->E ( use z4' ) ( use z2" ))
                                    (_ by &E
                                            ( embed bm3)
                                            ( assume bm31:sorted(m3),
                                                    bm32:perm(cons a m2,m3) in
                                                    embed bm32
                                              end )
                                    )
                            )

                    line 2 = perm(m3, cons a b) by ->E ( use z2' ) ( use z3' )

                    ? m:listof nat. sorted(m) & perm(m, cons a b)
                    by ?I m3 (_ by &I ( use 1 ) ( use 2 ))
                    end )
            end )
    end

    ? m:listof nat. sorted(m) & perm(m,l)
    by listE ( embed l ) ( use nilcase ) ( use conscase )
    end )
end
```

Figure 4.3.1 Input Proof for Sort

4.3.2 Output Programming Code for Sort

After passing the input proof of the function sort to the proof checker, realizer, and translator, it is converted into an insertion sort program. Figure 4.3.2a shows the actual output term from the translator. As in the previous section, for readability purposes, Figure 4.3.2b provides the edited version of the output by modifying some keywords and punctuations in the original output shown in Figure 4.3.2a. Figure 4.3.2c then shows an insertion sort function **isort** written in *Standard ML*. What we expect is that the output programming code for function **sort** matches the function **isort**.

Function **isort** states that it takes a list and sorts it by using function **insert** defined in Figure 4.2.2c as follows:

> if the input list is nil, then nil is returned since it is already sorted; or else a recursive call to **isort** is made with the tail of the list as an argument, and then **insert** is called with the sorted tail list and the head element as its arguments. Function **insert** inserts the head element into the proper place in the sorted tail list.

This is equivalent to the programming code in Figures 4.3.2a and 4.3.2b, where we have an input l by the lambda abstraction, and the list recursion on l. If l is nil, then the result itself is nil; or else if l is cons a b then the result is the list m3, which involves two substitutions: first m3 is substituted with the application of insert to the list m2 and an element a, where m2 is then substituted with a recursive call with the tail list b, that is, the head element a is inserted into the proper place in the sorted tail list.

```
lambda ("l",
   listrec ( exprt (id "l"),
             exprt nil,
             "a","b","rec_b",
                 let1("m2", ident "rec_b",
                       let1 ("m3", apply (apply (ident "insert", exprt (id "m2")),
                                          exprt (id "a")),
                             exprt (id "m3")))))
```

Figure 4.3.2a Translated Programming Code for Sort

```
lambda l. listrec l of
                 isnil. nill
                 (iscons a b).rec_b. let m2=rec_b in let m3=(insert m2) a in m3
```

Figure 4.3.2b Translated Programming Code for Sort (Edited)

```
fun isort [] = []
|    isort a::b = let val m2=isort b in
                        let val m3=insert m2 a in m3 end end ;
```

Figure 4.3.2c Function Sort Written in *Standard ML*

4.4 Linking of Insert and Sort

In the previous section, we have successfully translated the proofs of **insert** and **sort** into programs. As mentioned earlier, the proof for **sort** uses the function **insert** as an assumption from which to derive. Therefore, these two programs can be connected by using the cut rule. The resulting linked program is the complete insertion sort program. The linking process is done by using the linker, explained in the next sections.

4.4.1 Inputs

According to the cut rule in Figure 3.4.1, the linker requires five inputs, which are listed as follows:

1. term for insert - the output of the translator shown in Figure 4.2.2a,
2. sequent for insert - the output of the translator with the above term,
3. term for sort - the output of the translator shown in Figure 4.3.2a,
4. sequent for sort - the output of the translator with the above term, and
5. connecting point variable "insert," which is used in the context of the above sequent.

The file test.sml provides the function **testl**, which takes two input proof files and the connecting point variable, passes each proof to the proof checker, realizer, and translator, and then links the translated programs at the connecting point by calling the function **link** with the above list of five inputs. The output is the same as the output of the linker, that is, the linked term and new sequent, which are described in the next section.

4.4.2 Outputs

Figure 4.4.2a shows the actual output term from the linker. As in the previous sections, for readability purposes, Figure 4.4.2b provides the edited version of the output by modifying some keywords and punctuations in the original output shown in Figure 4.4.2a. The **lett** constructor in Figure 4.4.2a indicates what is substituted, in other words, where the two programs are linked. Figure 4.4.2c shows an insertion sort function **isort** written in *Standard ML*. What we expect is that the linked program for

the functions **insert** and **sort** shown in Figure 4.4.2a and Figure 4.4.2b matches the function **isort** to the definition of the function **insert** shown in Figure 4.4.2c.

The program in Figure 4.4.2c states that **insert** is a function to be used in the function **isort**, that is, all the occurrences of **insert** in **isort** are referred to as the defined function body of **insert**. The programming code in Figure 4.4.2a states to substitute the programming code of **insert** for all free occurrences of "insert" in the programming code of **sort**. Therefore, it can be said that the programming code in Figure 4.4.2a stands for the one in Figure 4.4.2c. Hence, we are assured that the linking of **insert** and **sort** is done successfully.

```
lett ("insert",
      lambda ("l",
         listrec (exprt (id "l"),
            lambda ("a",exprt (cons (id "a",nill))),
            "x","y","rec_y",
                  lambda ("a",
                      casee (lawt (or (predicate ("<",[id "a",id "x"]),
                                    nott (predicate ("<",[id "a",id "x"])))),
                           "p", exprt (cons (id "a",cons (id "x",id "y"))),
                           "q", let1 ("m2",apply (ident "rec_y",exprt (id "a")),
                                       exprt (cons (id "x",id "m2")))))))),
      lambda ("l", listrec (exprt (id "l"),
                           exprt nill,
                           "a","b","rec_b",
                                 let1 ("m2",ident "rec_b",
                                       let1 ("m3",
                                             apply (apply (ident "insert",
                                                           exprt (id "m2")),exprt (id "a")),
                                             exprt (id "m3"))))))
```

Figure 4.4.2a Linked Programming Code for insert and sort

```
let insert=lambda l.
                listrec l of
                        isnil lambda a. cons a nill
                       (iscons x y).rec_y.
                           lambda a. case (<(a, x) or not <(a, x)) of
                                          (inl p). cons a (cons x, y)
                                          (inr q). let m=rec_y a in cons x m2
in lambda l. listrec l of
                  isnil. nill
                 (iscons a b).rec_b. let m2=rec_b in let m3=(insert m2) a in m3
```

Figure 4.4.2b Linked Programming Code for insert and sort (Edited)

```
let fun insert [] a = [a]
    |    insert x::y a = if a<x then a :: x :: y
                                  else let val m2=insert y a in x :: m2 end
in  fun isort [] = []
    |    isort a::b = let val m2=isort b in
                        let val m3=insert m2 a in m3 end end
end ;
```

Figure 4.4.2c Function Insertion Sort Written in *Standard ML*

4.5 Example of a Run-Time Session

An input proof to our system needs to be saved into a file, which can be written by using available text editors such as *GNU Emacs* and *Visual.* As mentioned earlier, we are using a run-time system of the *Standard ML, SML/NJ (0.93).* Here is a demonstration of a sample run-time session of our system for the example proof saved in the file "test.example." The bold faced texts indicate what the user typed in. The first text **sml** starts the *Standard ML*, then type in the second text:

use "make.sml";

followed by <Return>, which opens the make file of our system to load and compile

all the files necessary. After loading the system, call the function run with the input file name by typing the third text:

run "test.example";

The function run takes an input file containing a proof and calls the functions parse, check, attach_polarity, and translate in this order to convert an input proof into a programming code.

```
% sml

Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit

- use "make.sml";
[opening make.sml]

. . .

val it = () : unit

- run "test.example";

Input is:

predicate P
predicate Q

assume a : P -> Q , b : P in
P & Q  by &I ( embed b ) ( Q by ->E ( embed a ) ( embed b ) )
end

EOF
EOF

val it =
(empty,
([("b",predicate ("P",[])),
  ("a",hook (predicate ("P",[]),predicate ("Q",[])))],
 andd (predicate ("P",[]),predicate ("Q",[]))) : term * sequent
-
```

Chapter 5
Conclusion

This chapter gives a summary of this report, and suggests possible future work.

5.1 Summary

Writing a correct programming code is necessary in computer system development. By studying intuitionistic logic together with type theory, we can come up with the important correspondences:

proofs are programs are values, and

propositions are types are specifications [16].

Based on the intuitionistic idea, these correspondences lead to the development of correct programming code, and realizability theory states how it can be accomplished. By using Hatcliff's algorithm for realizability theory, we have designed a proof language and implemented a system which translates logical proofs written in the language into correct programming code.

5.2 Future Work

The system can be improved in many possible ways. Some of the possibilities include simplification of the proof language grammar, additional features to the system, and extended case studies, each of which is discussed below.

Simplification of the Grammar

After the case study in Chapter 4, we immediately notice that a proof written in our proof language tends to be large, especially as a proof gets larger and more complicated. Although we have some simplification features in the grammar described in Chapter 3, proofs that can be used for usual programming development, such as an insertion sort program, are still long. It is obviously discouraging for programmers to write such long code for a simple program like a sorting program, while they can

write a much shorter program in other programming languages. Therefore, simplification of the proof language will be necessary. Some suggestions are listed as follows:

• Assignment Feature

An assignment feature, which assigns a name to some part of a proof such as propositions and sequent, can be introduced in the grammar. Then, the repetitions can be reduced in a proof. The current system provides the abstraction only for proofs. The type checking process needs to be modified to deal with scopes.

• Universal Elimination Rule

In order to utilize an axiom with several nested universal quantifiers, the universal elimination rule needs to be used multiple times to remove the quantifiers one at a time. It is somewhat redundant, and if more than one universal quantifier can be eliminated by applying corresponding values at once, it will speed up the process and reduce proof steps.

Additional Features

Some suggestions for additions to the system are listed as follows:

• Nat Elimination Rule

Instead of the list elimination rule, Hatcliff's original algorithm includes mathematical induction, that is, the elimination rule for natural numbers. The elimination rule states the mathematical induction principle and works quite similarly to the list elimination rule. With the nat elimination rules, more mathematical properties and functions can be derived.

• Comment Feature

The current system allows writing only a proof in an input file. It would be convenient if the user could put some comments in the input file along with the proof.

• System Interface and Pretty Printing

There is no user-friendly interface to our system nor a "pretty printing" feature for the output. For instance, as the proof checker described in [10], the *GNU Emacs* editor can be used as a real time editor of the proof checker so that the user can easily edit an input file, submit it to the run-time environment, and also see the output within the editor. Moreover, if a correction needs to be made to the input, it can be done easily within the editor as well. The "pretty printing" function can be added to the system so that the output is formatted to be more readable.

• Condition for the Linker

By allowing variables with the same name, but associated with different propositions in two contexts, the third condition of the linking process described in Section 3.4.2 can be relaxed, and the process would become more flexible. In order to avoid "type crash" in the merged context, one of the variable names needs to be changed prior to the merge, which may take some work in the linker.

Case Studies for Other Sorting Algorithms

Different sorting specifications for selection, merge, bubble, and quick sorts can be developed and studied as in the insertion sort in Chapter 4.

Bibliography

[1] Aho, A., Sethi, R., and Ullman, J. Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, MA, 1986.

[2] van Dalen, D. Brouwer's Cambridge Lectures on Intuitionism. Cambridge University Press, Cambridge, 1981.

[3] van Dalen, D. Logic and Structure. 2d. ed. Springer-Verlag, Berlin, 1983.

[4] DeSouza, J. Conversion of Logic Proofs to Computer Programs. Department of Computing and Information Sciences, Kansas State University, 1993.

[5] Dragalin, A. G. Mathematical Intuitionism: Introduction to Proof Theory. American Mathematical Society, Providence, Rhode Island, 1988.

[6] Dummett, M. Elements of Intuitionism. Oxford University Press, Oxford, 1977.

[7] Girard, J.-Y., Lafont, Y., and Taylor, P. Proofs and Types. Cambridge University Press, Cambridge, 1989.

[8] Gunter, Carl A. Semantics of Programming Languages. MIT Press, Cambridge, 1992.

[9] Hankley, William. Program Specification and Verification. Department of Computing and Information Sciences, Kansas State University, 1991.

[10] Karjian, N. A. A Proof Checker for Propositional and Predicate Logic. Department of Computing and Information Sciences, Kansan State University, 1994.

[11] Kleene, S. On the Interpretation of Intuitionistic Number Theory. Journal Symbol Logic 10, 1945, 109-24.

[12] Kleene, S. Introduction to Metamathematics, North-Holland, Amsterdam, 1952.

[13] Martin-Löf, P. Intuitionistic Type Theory. Bibliopolis, Naples, 1984.

[14] Martin-Löf, P. Constructive Mathematics and Computer Programming. In C. A. R. Hoare, ed. Mathematical Logic and Programming Languages. Prentice-Hall, Englewood Cliffs, NJ, 1985, 167-84.

[15] Mogan, Carroll. Programming from Specifications. Prentice Hall, Englewood Cliffs, NJ, 1990.

[16] Schmidt, D. A. The Structure of Typed Programming Languages. MIT Press, Cambridge, 1994.

[17] Troelstra, A. S. Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Springer-Verlag, Berlin, 1973.

[18] Troelstra, A. S. and van Dalen, D. Constructivism in Mathematics: An Introduction. vol. I. North-Holland, Amsterdam, 1988.

[19] Turner, R. Constructive Foundations for Functional Languages. McGraw Hill, New York, 1991.

Appendix A
Source Files

The source files used in our system, with possible future modifications, are available via ftp at ftp.cis.ksu.edu directory

/pub/CIS/students/realizer

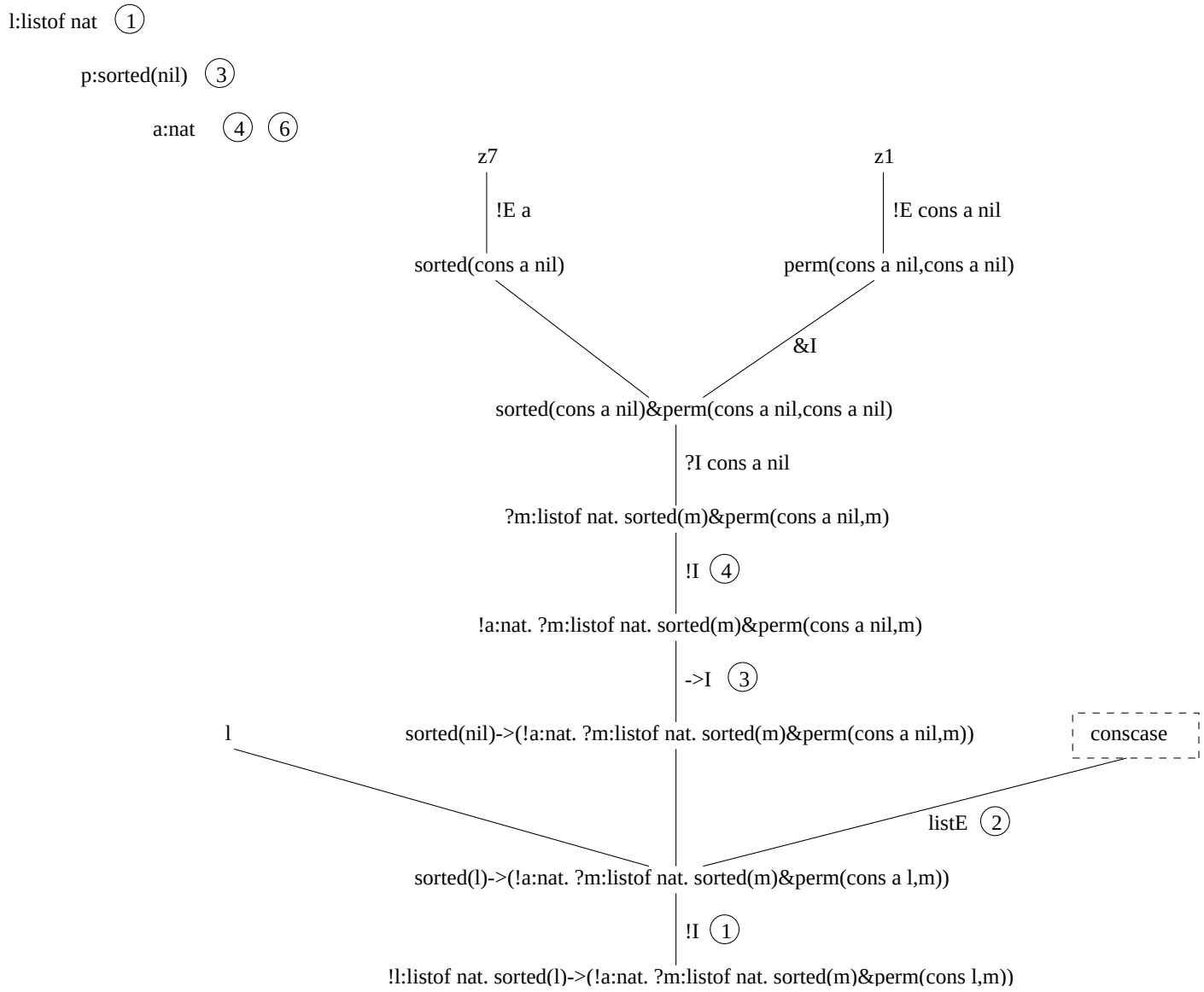
or at the URL

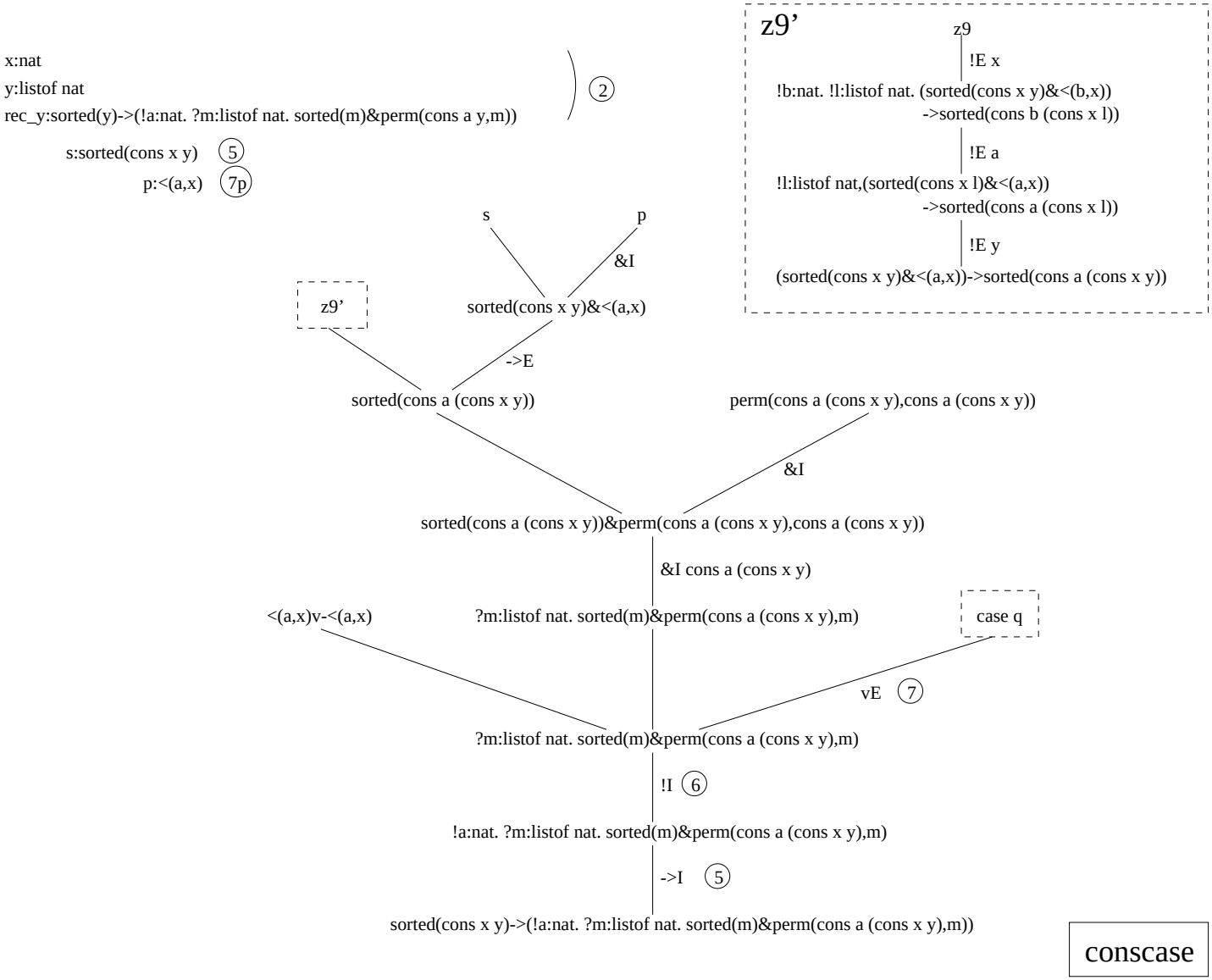
file://ftp.cis.ksu.edu/pub/CIS/students/realizer

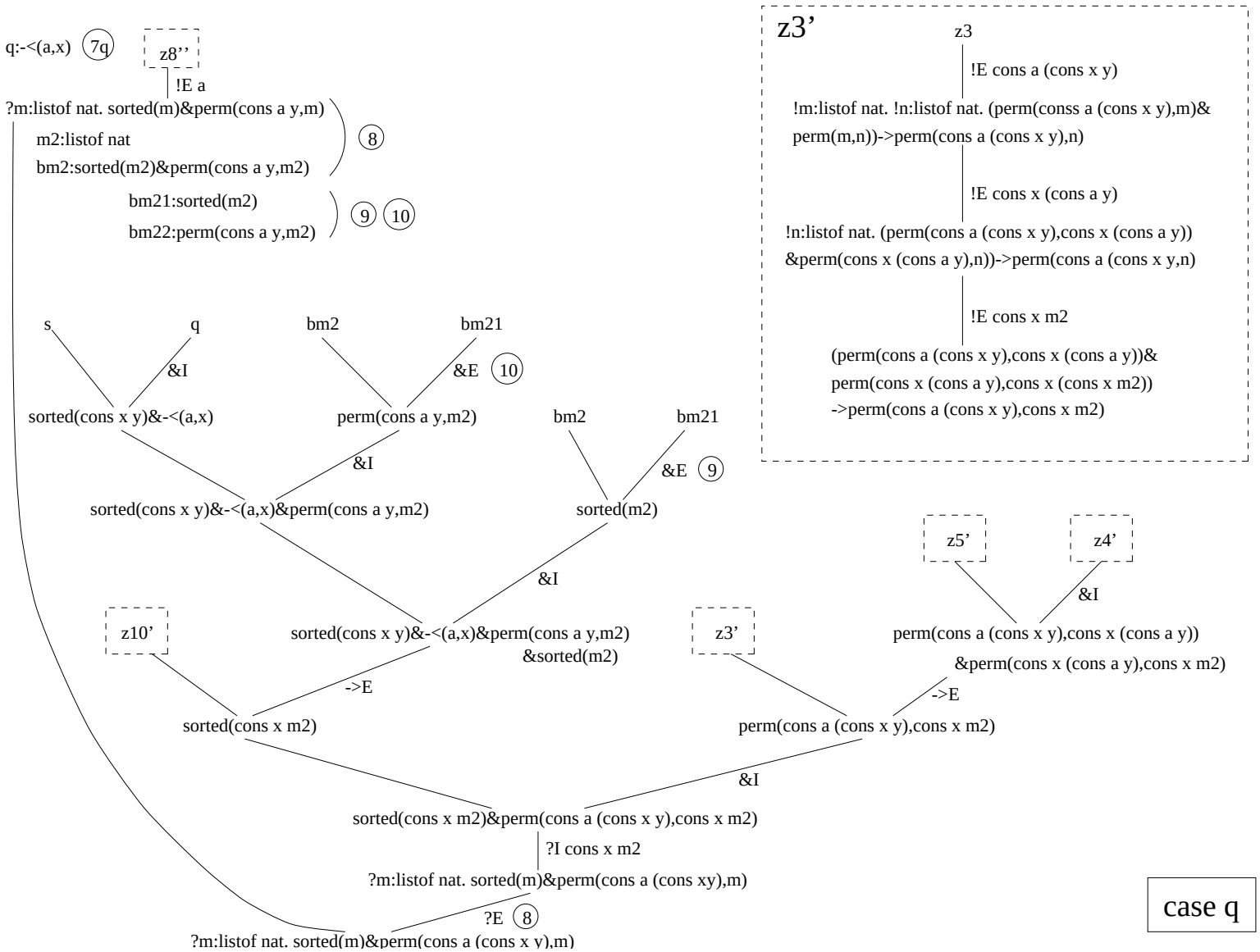
in the Department of Computing and Information Sciences, Kansas State University.

Appendix B
Natural Deduction Tree for Function Insert

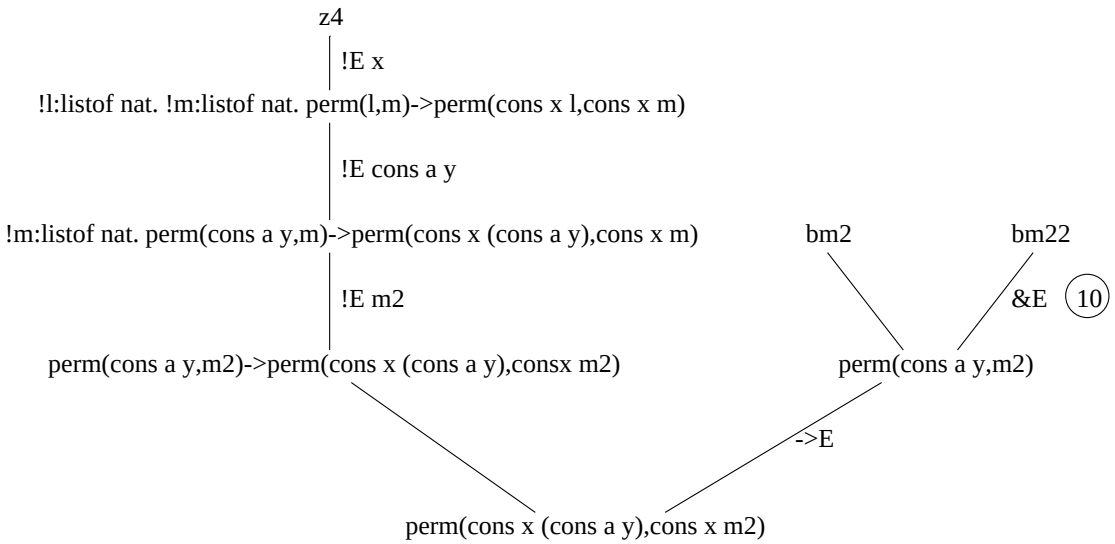
The natural deduction tree for the proof of the function **insert**, which is discussed in Section 4.2, is provided in the following pages. As described in Chapter 2, the leaves are axioms and the root is the desired proposition. The number in a circle, which is placed next to an inference rule and its local assumption, indicates a scope of the local assumption. The leaf in a dotted rectangle is expanded further, and most of the time, the name of the leaf is matched to the corresponding proof abstraction found in Figure 4.2.1.


l:listof nat 1
p:sorted(nil) 3
a:nat 4 6
z7
z1
!E a
!E cons a nil
sorted(cons a nil)
perm(cons a nil,cons a nil)
&I
sorted(cons a nil)&perm(cons a nil,cons a nil)
?I cons a nil
?m:listof nat. sorted(m)&perm(cons a nil,m)
!I 4
!a:nat. ?m:listof nat. sorted(m)&perm(cons a nil,m)
->I 3
l
sorted(nil)->(!a:nat. ?m:listof nat. sorted(m)&perm(cons a nil,m))
conscase
listE 2
sorted(l)->(!a:nat. ?m:listof nat. sorted(m)&perm(cons a l,m))
!I 1
!l:listof nat. sorted(l)->(!a:nat. ?m:listof nat. sorted(m)&perm(cons l,m))

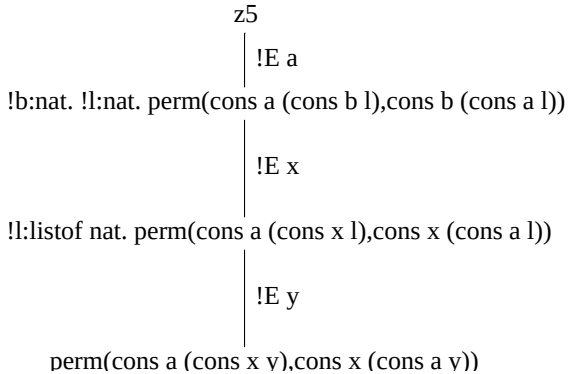

x:nat
y:listof nat
rec_y:sorted(y)->(!a:nat. ?m:listof nat. sorted(m)&perm(cons a y,m))
2
s:sorted(cons x y) 5
p:<(a,x) 7p
z9'
z9
!E x
!b:nat. !l:listof nat. (sorted(cons x y)&<(b,x))
->sorted(cons b (cons x l))
!E a
!l:listof nat,(sorted(cons x l)&<(a,x))
->sorted(cons a (cons x l))
!E y
(sorted(cons x y)&<(a,x))->sorted(cons a (cons x y))
s
p
&I
z9'
sorted(cons x y)&<(a,x)
->E
sorted(cons a (cons x y))
perm(cons a (cons x y),cons a (cons x y))
&I
sorted(cons a (cons x y))&perm(cons a (cons x y),cons a (cons x y))
&I cons a (cons x y)
<(a,x)v-<(a,x)
?m:listof nat. sorted(m)&perm(cons a (cons x y),m)
case q
vE 7
?m:listof nat. sorted(m)&perm(cons a (cons x y),m)
!I 6
!a:nat. ?m:listof nat. sorted(m)&perm(cons a (cons x y),m)
->I 5
sorted(cons x y)->(!a:nat. ?m:listof nat. sorted(m)&perm(cons a (cons x y),m))
conscase


q:-<(a,x) 7q
z8''
!E a
?m:listof nat. sorted(m)&perm(cons a y,m)
m2:listof nat
bm2:sorted(m2)&perm(cons a y,m2)
8
bm21:sorted(m2)
bm22:perm(cons a y,m2)
9 10
z3'
z3
!E cons a (cons x y)
!m:listof nat. !n:listof nat. (perm(conss a (cons x y),m)&
perm(m,n))->perm(cons a (cons x y),n)
!E cons x (cons a y)
!n:listof nat. (perm(cons a (cons x y),cons x (cons a y))
&perm(cons x (cons a y),n))->perm(cons a (cons x y,n)
!E cons x m2
(perm(cons a (cons x y),cons x (cons a y))&
perm(cons x (cons a y),cons x (cons x m2))
->perm(cons a (cons x y),cons x m2)
s
q
&I
sorted(cons x y)&-<(a,x)
bm2
bm21
&E 10
perm(cons a y,m2)
&I
sorted(cons x y)&-<(a,x)&perm(cons a y,m2)
bm2
bm21
&E 9
sorted(m2)
&I
sorted(cons x y)&-<(a,x)&perm(cons a y,m2)
&sorted(m2)
z10'
->E
sorted(cons x m2)
z5'
z4'
&I
perm(cons a (cons x y),cons x (cons a y))
&perm(cons x (cons a y),cons x m2)
z3'
->E
perm(cons a (cons x y),cons x m2)
&I
sorted(cons x m2)&perm(cons a (cons x y),cons x m2)
?I cons x m2
?m:listof nat. sorted(m)&perm(cons a (cons xy),m)
?E 8
?m:listof nat. sorted(m)&perm(cons a (cons x y),m)
case q

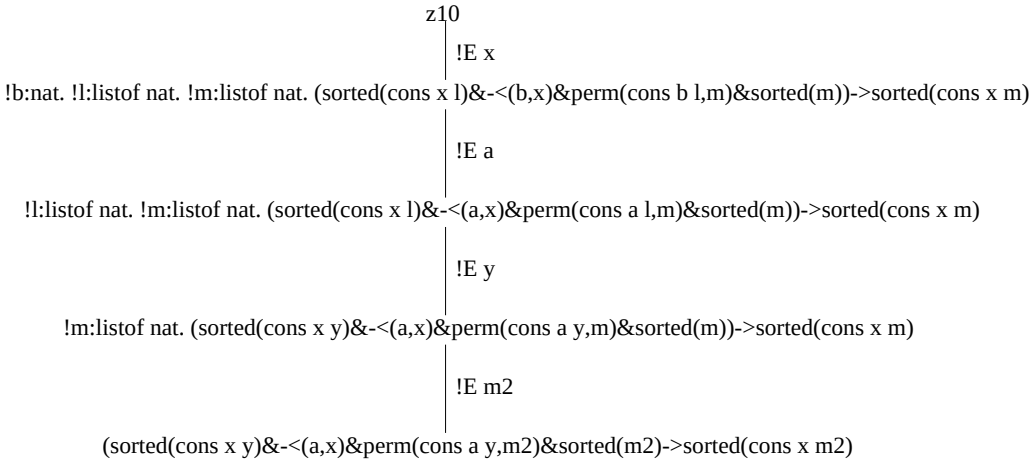
z4’


z4
!E x
!l:listof nat. !m:listof nat. perm(l,m)->perm(cons x l,cons x m)
!E cons a y
!m:listof nat. perm(cons a y,m)->perm(cons x (cons a y),cons x m)
bm2
bm22
!E m2
&E
10
perm(cons a y,m2)->perm(cons x (cons a y),consx m2)
perm(cons a y,m2)
->E
perm(cons x (cons a y),cons x m2)

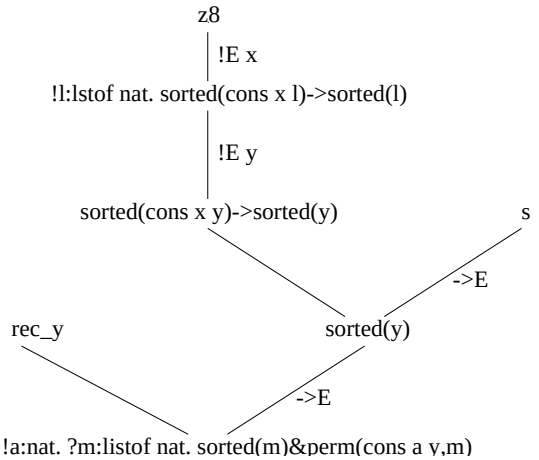
z5’


z5
!E a
!b:nat. !l:nat. perm(cons a (cons b l),cons b (cons a l))
!E x
!l:listof nat. perm(cons a (cons x l),cons x (cons a l))
!E y
perm(cons a (cons x y),cons x (cons a y))

z10’

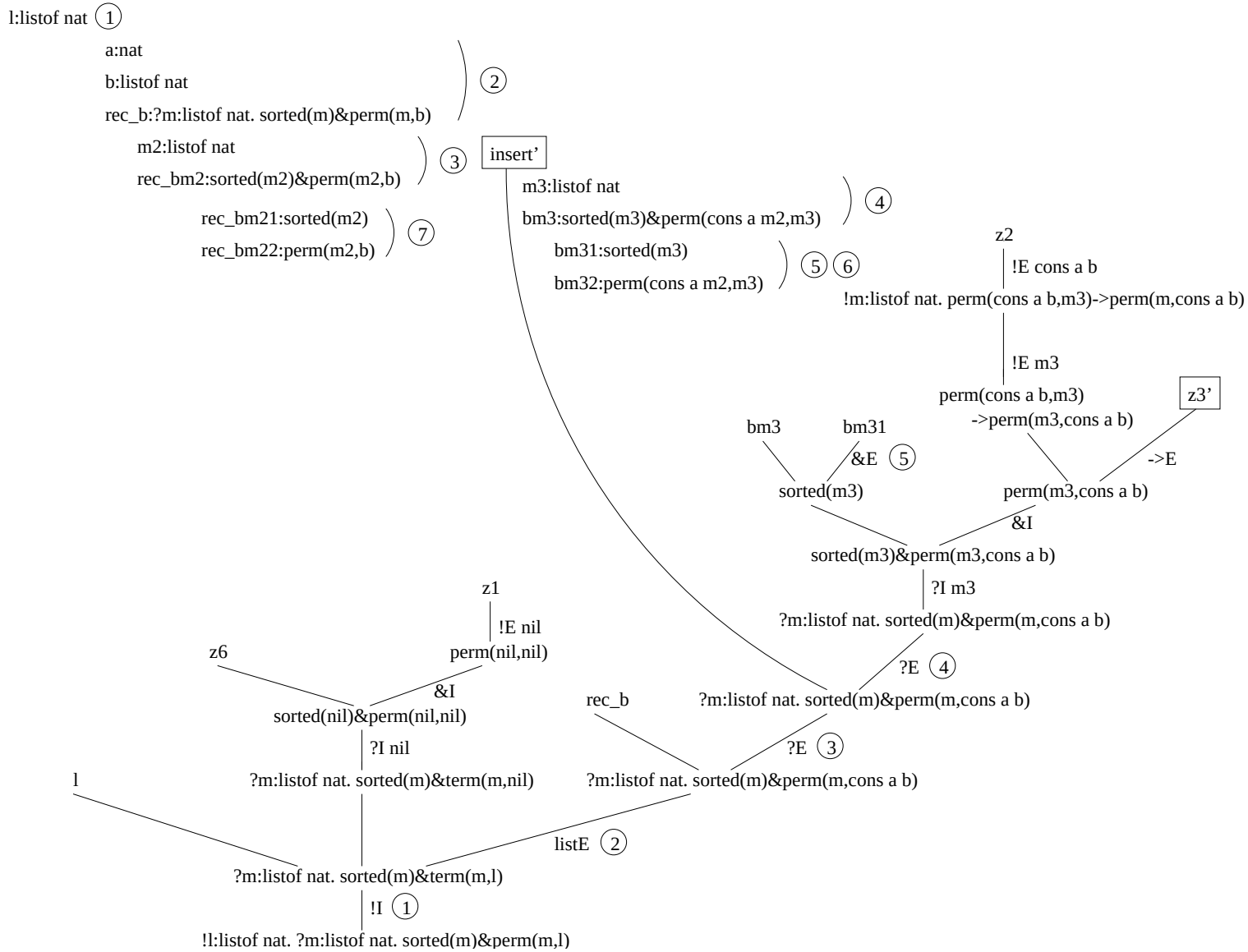

z10
!E x
!b:nat. !l:listof nat. !m:listof nat. (sorted(cons x l)&-<(b,x)&perm(cons b l,m)&sorted(m))->sorted(cons x m)
!E a
!l:listof nat. !m:listof nat. (sorted(cons x l)&-<(a,x)&perm(cons a l,m)&sorted(m))->sorted(cons x m)
!E y
!m:listof nat. (sorted(cons x y)&-<(a,x)&perm(cons a y,m)&sorted(m))->sorted(cons x m)
!E m2
(sorted(cons x y)&-<(a,x)&perm(cons a y,m2)&sorted(m2)->sorted(cons x m2)

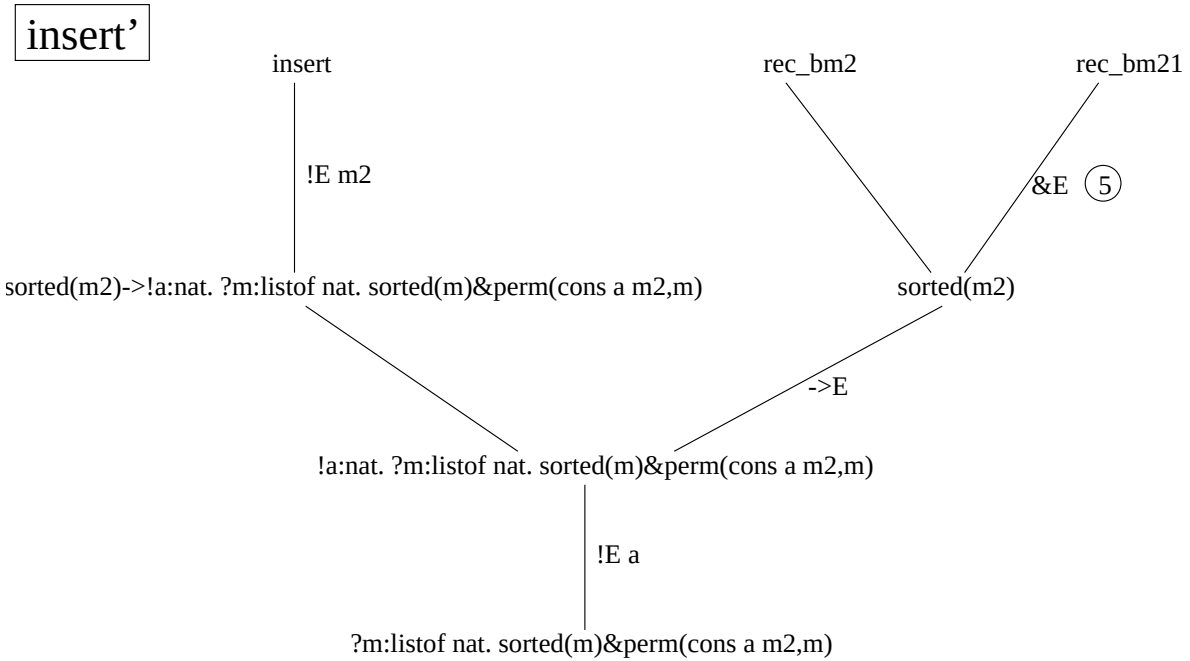
z8’’

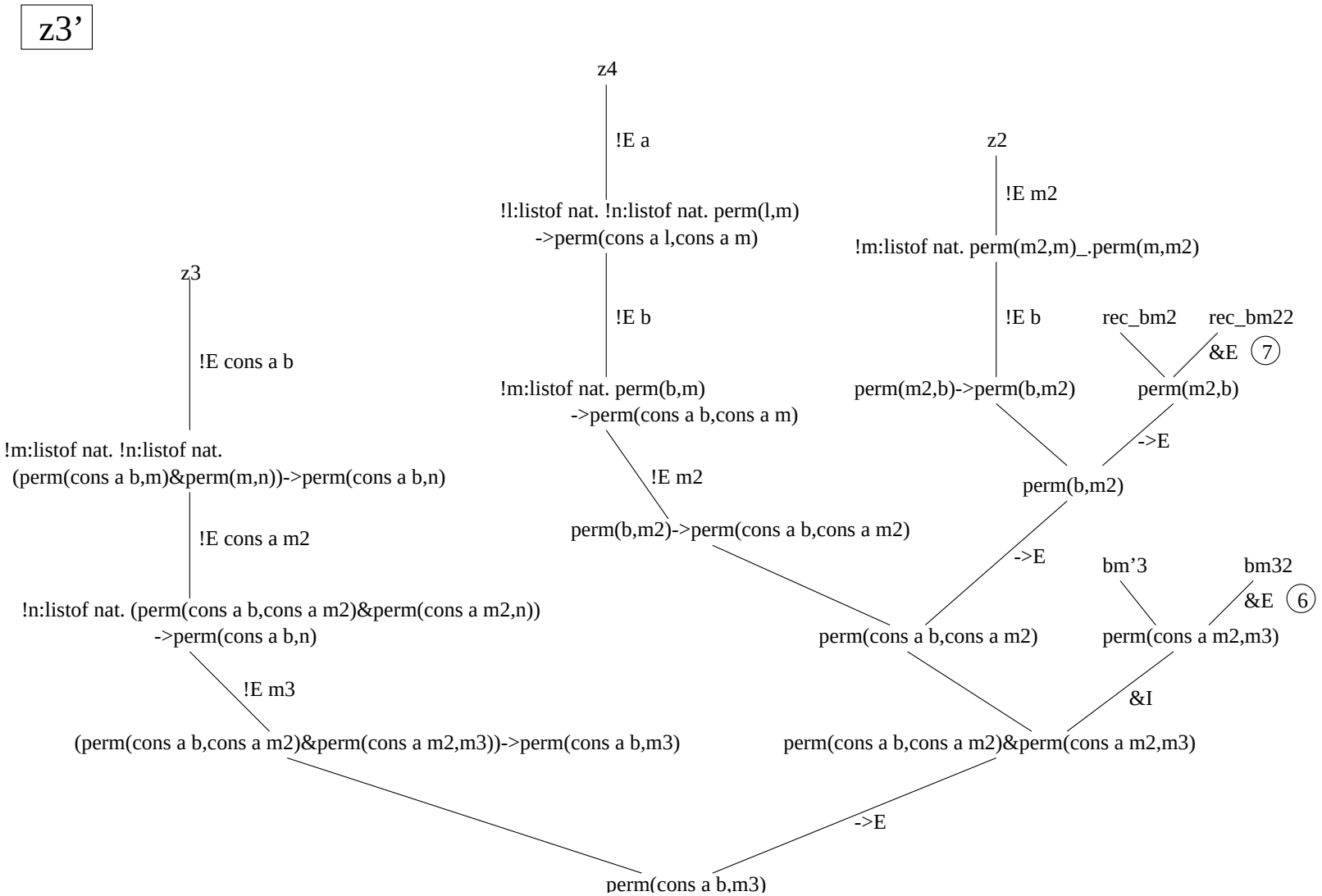

z8
!E x
!l:lstof nat. sorted(cons x l)->sorted(l)
!E y
sorted(cons x y)->sorted(y)
s
->E
rec_y
sorted(y)
->E
!a:nat. ?m:listof nat. sorted(m)&perm(cons a y,m)

Appendix C
Natural Deduction Tree for Function Sort

The natural deduction tree for the proof of the function **sort**, which is discussed in Section 4.3, is provided in the following pages. As described in Chapter 2, the leaves are axioms and the root is the desired proposition. The number in a circle, which is placed next to an inference rule and its local assumption, indicates a scope of the local assumption. The leaf in a dotted rectangle is expanded further, and the name of the leaf is matched to the corresponding proof abstraction found in Figure 4.3.1.


l:listof nat 1
a:nat
b:listof nat
rec_b:?m:listof nat. sorted(m)&perm(m,b)
2
m2:listof nat
rec_bm2:sorted(m2)&perm(m2,b)
3
insert'
m3:listof nat
bm3:sorted(m3)&perm(cons a m2,m3)
4
rec_bm21:sorted(m2)
rec_bm22:perm(m2,b)
7
bm31:sorted(m3)
bm32:perm(cons a m2,m3)
5 6
z2
!E cons a b
!m:listof nat. perm(cons a b,m3)->perm(m,cons a b)
!E m3
perm(cons a b,m3)
->perm(m3,cons a b)
z3'
bm3
bm31
&E 5
->E
sorted(m3)
perm(m3,cons a b)
&I
sorted(m3)&perm(m3,cons a b)
?I m3
?m:listof nat. sorted(m)&perm(m,cons a b)
z1
!E nil
z6
perm(nil,nil)
?E 4
&I
rec_b
?m:listof nat. sorted(m)&perm(m,cons a b)
sorted(nil)&perm(nil,nil)
?I nil
?E 3
l
?m:listof nat. sorted(m)&term(m,nil)
?m:listof nat. sorted(m)&perm(m,cons a b)
listE 2
?m:listof nat. sorted(m)&term(m,l)
!I 1
!l:listof nat. ?m:listof nat. sorted(m)&perm(m,l)


insert'
insert
rec_bm2
rec_bm21
!E m2
&E
5
sorted(m2)->!a:nat. ?m:listof nat. sorted(m)&perm(cons a m2,m)
sorted(m2)
->E
!a:nat. ?m:listof nat. sorted(m)&perm(cons a m2,m)
!E a
?m:listof nat. sorted(m)&perm(cons a m2,m)


z3’
z4
!E a
!l:listof nat. !n:listof nat. perm(l,m)
->perm(cons a l,cons a m)
!E b
!m:listof nat. perm(b,m)
->perm(cons a b,cons a m)
!E m2
perm(b,m2)->perm(cons a b,cons a m2)
z2
!E m2
!m:listof nat. perm(m2,m)_.perm(m,m2)
!E b
perm(m2,b)->perm(b,m2)
rec_bm2
rec_bm22
&E 7
perm(m2,b)
->E
perm(b,m2)
->E
bm’3
bm32
&E 6
perm(cons a b,cons a m2)
perm(cons a m2,m3)
&I
perm(cons a b,cons a m2)&perm(cons a m2,m3)
->E
perm(cons a b,m3)
z3
!E cons a b
!m:listof nat. !n:listof nat.
(perm(cons a b,m)&perm(m,n))->perm(cons a b,n)
!E cons a m2
!n:listof nat. (perm(cons a b,cons a m2)&perm(cons a m2,n))
->perm(cons a b,n)
!E m3
(perm(cons a b,cons a m2)&perm(cons a m2,m3))->perm(cons a b,m3)