FIFTEEN SIMPLE EXERCISES IN

HIGHER-ORDER INTUITIONISTIC LOGIC

by

Jon Shultis

CU-CS-307-85 July, 1985

University of Colorado, Department of Computer Science,
Boulder, Colorado.

Fifteen Simple Exercises in Higher-Order Intuitionistic Logic

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

Abstract

Intuit 1.1 is a computer system for deriving programs as byproducts of mathematical proofs. The proofs are conducted in a higher-order intuitionistic logic. Since the formalism of the **Intuit** logic is likely to be unfamiliar to most computer scientists, fifteen elementary exercises in the logic are collected in this report, together with their solutions. The introduction of derived inference rules into the system is also discussed, and several such rules are developed in the course of solving the exercises.

Fifteen Simple Exercises in Higher-Order Intuitionistic Logic

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

The purpose of these exercises is to introduce the reader to the art of deriving computer programs by proving mathematical theorems. Because any skill is best learned by working examples, this report has been structured as a workbook. After some introductory remarks, we present, in **§2**, a summary of **Intuit** [9], [10] the "programming language" to be used in the exercises listed in **§3**. Solutions to the exercises are given in **§4**.

1. Introduction

Writing correct programs is difficult in part because of the conceptual distance between the text of a program and its purpose. Witness the novice programmer explaining a program by tracing its execution: "...then, if x is greater than 1, it assigns 5 to sum, then does the loop again...". Nowhere in such an "explanation" can one find any indication of *why* these things are being done. Although the programmer may have some intuitive understanding of how the program accomplishes its purpose, the formal articulation of that intuition is likely to be prohibitively difficult and boring, and hence pointless [3].

Although most programs may make boring theorems, good theorems often make interesting programs. For example, the theorem that for every deterministic finite automaton there is a minimum-state equivalent automaton leads to a program having applications in such diverse areas as compilers and VLSI design. So, though mathematics may not be a panacea for incorrect software, it is one important source of software.

Programs can be automatically extracted from proofs of mathematical theorems, using intuitionistic logic. Prior systems that exploit this fact include Martin-Löf's Constructive Set Theory [6], [8], and the program refinement logics (prls) of Constable *et al.* [1], [2]. The **Intuit** systems are based on a different logic, one that is more suitable for "ordinary" mathematics, with special attention to topos theory.

The proviso that theorems be proved using a (formal) intuitionistic logic is important; it is the logic that makes the automatic derivation of software from mathematics possible. This situation contrasts sharply with the main difficulty facing program verification in general, *viz.* that it is essentially impossible to extract an appropriate theorem (or proof) from the text of a program.

One may reasonably wonder whether formal intuitionistic proofs could ever be involved in the day-to-day social process of "real" mathematics. The growing community of mathematicians actively pursuing the intuitionistic reformulation of mathematics makes it unnecessary to become embroiled in a philosophical debate about intuitionism; the only real issue is whether the amount of detail required for a complete formal proof is prohibitive. De Millo *et al.* [3] argue that formal derivations are necessarily too lengthy to be practical, basing their argument on studies of the computational complexity of decision problems in formal theories [11], [7]. These studies, however, are largely irrelevant. They ignore the use of earlier results as lemmas, and the use of generalization to collapse many specific results into one pattern. They do not consider the condensation possible from the use of higher-order abstractions. They make no allowance for the introduction of derived inference rules, formula matching, and other labor-saving devices. Finally, the results on derivation length are for decision algorithms, which lack the imagi-

nation (to say nothing of the parsimoniousness) of human mathematicians. When these things are included in the analysis, it becomes much less clear that the task of devising a detailed proof is intrinsically more complex than that of writing and debugging a Pascal program for the same task. When we consider that the program derived from the proof is more likely to be correct, filling in the details of a proof may seem like quite a bargain.

2. Summary of Intuit 1.1

The fifteen exercises collected here represent the most rudimentary kinds of results that can be obtained with the **Intuit** system (version 1.1). **Intuit** is based on an axiomatization of higher-order intuitionistic logic that is quite small, by programming language standards. That is, the axiomatization is more like pure Lisp than Interlisp in its complexity and sophistication, and the examples are correspondingly simple.

Intuit consists of a set of axiom schemata and inference rules. The notions of theorem and proof, and proof from hypotheses, are defined as usual. So, for example, the result of each axiom is an object called a "theorem", and the inference rules combine theorems and formulae to produce new theorems.

Each theorem has three parts: a (possibly empty) set of hypotheses, a formula called the *conclusion* of the theorem, and a λ-expression representing a program which "realizes" the theorem. A theorem with an empty set of hypotheses is called a *proposition*, and the program of a proposition is always a closed λ-term (i.e., contains no free variables).

Since the purpose of these exercises is to foster proficiency with the logic, however, we ignore the program component of theorems from now on. After all, the program is produced automatically as a by-product of a proof, and is not

germane to the art of proving. This may be somewhat disconcerting at first, because ordinarily we do the exact opposite: we write programs, ignoring their correctness proofs! For the present, we ask the reader to take it on faith that an appropriate program is being produced silently and to concentrate on the mathematics.

In what follows, f g and h denote arbitrary formulae, x y and z are variables, and τ and σ denote arbitrary terms. There is one unary primitive predicate **E**, such that $\mathbf{E}\tau$ effectively asserts that τ *exists*, and one binary primitive predicate $\equiv$, such that $\tau \equiv \sigma$ effectively asserts that τ and σ are equivalent terms. Terms in **Intuit** are sorted, but we shall only have occasion to use the sort Ω of truth values. For simplicitly, the reader may therefore assume that all terms are of this sort. The term !x.f is read as "the unique x satisfying f". Note that, like other terms, !x.f may or may not exist.

The axiom schemata are listed below, grouped according to whether they are propositional, first-order, or higher-order. For each schema we list on one line the notation recognized by the **Intuit** system, and the usual mathematical notation on the following line. In the **Intuit** notation, each schema takes as arguments an arbitrary set *hyp* of formulae to be used as hypotheses, and a tuple of terms and formulae used to instantiate it.

2.1. Propositional Axioms

K hyp (f,g)
hyp $\vdash$ f $\rightarrow$ (g $\rightarrow$ f)

S hyp (f,g,h)
hyp $\vdash$ (f $\rightarrow$ (g $\rightarrow$ h)) $\rightarrow$ ((f $\rightarrow$ g) $\rightarrow$ (f $\rightarrow$ h))

p1 hyp (f,g)
hyp $\vdash f \wedge g \rightarrow f$

p2 hyp (f,g)
hyp $\vdash f \wedge g \rightarrow g$

pair hyp (f,g)
hyp $\vdash f \rightarrow (g \rightarrow f \wedge g)$

Assume hyp f
where hyp is any set of formulae which includes f
hyp $\vdash f$

2.2. First-Order Axioms

substitutivity hyp (f,x,y,z)
hyp $\vdash (f[y/x] \wedge y \equiv z) \rightarrow f[z/x]$

extensionality hyp (x,y,z)
hyp $\vdash (\forall x.\ x \equiv y \leftrightarrow x \equiv z) \rightarrow y \equiv z$

instantiation hyp (x,f)
hyp $\vdash (\forall x.f \wedge \mathbf{E}x) \rightarrow f$

description hyp (y,x,f)
$\forall y.\ (y \equiv !x.f) \leftrightarrow (\forall x.\ f \leftrightarrow x \equiv y)$

2.3. Higher-Order Axiom

comprehension hyp (y,f,v1, ... , vn)
hyp $\vdash \mathbf{E}!y.\ \forall v1.\ \ldots \forall vn.\ f \leftrightarrow y(v1, \ldots, vn)$

The list of variables v1, ... , vn may be empty, in which case y essentially represents the characteristic function of the formula f.

2.4. Inference Rules

The inference rules are listed below. As before, the **Intuit** notation is shown first, but because this notation conceals the structure of the parameters to infer-

ence rules, the **Intuit** notation is followed by a note of the form "where ... " indicating the structure of the parameters, along with any restrictions on them.

Curry t
where t is a theorem of the form hyp $\vdash$ g$\wedge$**E**x $\rightarrow$ f
and x does not occur free in g.

$$\frac{\text{hyp} \vdash g \wedge \mathbf{E}x \rightarrow f}{\text{hyp} \vdash g \rightarrow \forall x.\, f}$$

MP (major, minor)
where major is a theorem of the form hyp $\vdash$ f $\rightarrow$ g
and minor is a theorem of the form hyp $\vdash$ f

$$\frac{\text{hyp} \vdash f \rightarrow g \qquad \text{hyp} \vdash f}{\text{hyp} \vdash g}$$

Sub (τ,x,thm)
where thm is a theorem of the form hyp $\vdash$ f,
and x does not occur free in any of the formulae in hyp

$$\frac{\text{hyp} \vdash f}{\text{hyp} \vdash f[\tau/x]}$$

elaborate ([f1; ... ; fn], [g1; ... ; gn], t)
where f1, ... , fn are arbitrary formulae,
g1, ... , gn are propositional variables,
t is a theorem of the form hyp $\vdash$ f,
and none of the g's occurs in any of the formulae in hyp.

$$\frac{\text{hyp} \vdash f}{\text{hyp} \vdash f[f1, \ldots, fn/\ g1, \ldots, gn]}$$

Invoke (t,f)
where t is any theorem of the form hyp $\vdash$ g,
whose hypotheses include f

$$\frac{\text{hyp} \vdash g}{\text{hyp} \vdash f}$$

discharge (t,f)
where t is a theorem of the form $hyp \vdash g$,
whose hypotheses include f.

$$\frac{hyp \vdash g}{hyp - \{f\} \vdash f \rightarrow g}$$

AddHyp (hypa,t)
where t is any theorem of the form $hypb \vdash f$

$$\frac{hypb \vdash f}{hypa \cup hypb \vdash f}$$

3. The Exercises

The following definitions will be used to abbreviate formulae.

$f \leftrightarrow g \triangleq (f \rightarrow g) \wedge (g \rightarrow f)$

$f \vee g \triangleq \forall z.\ ((f \rightarrow z()) \wedge (g \rightarrow z())) \rightarrow z()$

$\neg f \triangleq \forall z.\ f \rightarrow z()$

$\exists x.f \triangleq \forall z.\ (\forall x.\ f \rightarrow z()) \rightarrow z()$

$\top \triangleq E!y.y()$

$\bot \triangleq \forall z.z()$

With these definitions, prove each of the following.

1. $\vdash f \rightarrow f$
2. $\vdash f \leftrightarrow f$
3. $\vdash x \equiv x$
4. $\vdash (f \rightarrow g) \rightarrow ((g \rightarrow h) \rightarrow (f \rightarrow h))$
5. $\vdash g \leftrightarrow (!y.\ g \leftrightarrow y())()$
6. $\vdash f \rightarrow (f \vee g)$
7. $\vdash g \rightarrow (f \vee g)$

8. $\vdash (f \to g) \to ((h \to g) \to ((f \vee h) \to g))$

9. $\vdash (f \to h) \to ((g \wedge f) \to h)$

10. $\vdash x \equiv y \to y \equiv x$

11. $\vdash (f \to g) \to ((f \to \neg g) \to \neg f)$

12. $\vdash \neg f \to (f \to g)$

13. $\vdash (f \wedge \mathbf{E}x) \to \exists x.\, f$

14. $\vdash \bot \to f$

15. $\vdash \top$

4. Solutions

The theorem resulting from each of the following proofs is given a name, for convenient reference. The name of a theorem is the label attached to the last step in its proof.

4.1. id

Using the notation of **Intuit**, the solution to problem 1 is as follows.

```
step1: Assume({f}, f)
id:    Discharge(step1, f)
```

The translation of this into conventional notation yields the following.

```
step1: f ⊢ f          (by assumption)
id:    ⊢ f → f        (by deduction)
```

The solutions to the remaining exercises are given using only the **Intuit** notation. This not only encourages the reader to become familiar with the notation; it makes it a bit harder to "peek" at the answers! Incidentally, **Intuit** 1.1 includes a function called "theorem" which displays a theorem in conventional mathematical notation at the user's request. Future versions of the system should give this kind of feedback automatically.

4.2. reflex↔

```
step1: pair((f → f), (f → f))
step2: MP(step1, id)
reflex↔: MP(step2, id)
```

The technique, illustrated by this proof, of forming a conjunction from a pair of theorems, is so common that it behooves us to formalize it as a derived inference rule.

conjoin (t1, t2)

$$\frac{\vdash t1 \quad \vdash t2}{\vdash t1 \wedge t2}$$

Since **Intuit** 1.1 is written in ml [5], it is easy for the **Intuit** programmer to define new inference rules, such as conjoin, by composing old ones. By way of illustration, the ml code for conjoin is as follows. destthm is a function which breaks a theorem into its three parts: hypotheses, conclusion, and program.

```
let conjoin (t1, t2) = let (h1, c1, p1) = destthm t1
                       and (h2, c2, p2) = destthm t2 in
                         MP( MP( pair h1 (c1, c2), t1), t2)
```

Henceforth we will introduce derived inference rules whenever it seems appropriate to do so, without detailing the ml code.

4.3. reflex≡

step1: elaborate ([x≡y], [f], reflex↔)
step2: K ((x≡y ↔ x≡y), ((f → f) ∧ (**E**x)))
step3: MP(step2, step1)
step4: Curry step3
step5: MP(step4, id)
step6: extensionality {} (x,y,y)
reflex≡: MP(step6, step5)

We can extract two derived inference rules from patterns appearing in the preceding proof. The purpose of steps 2 and 3 is to make the conclusion of a theorem into the consequent of an implication with an arbitrary antecedent. This leads to the derived inference rule *antecede*.

antecede (g, t)
where t is a theorem of the form hyp ⊢ f

$$\frac{hyp \vdash f}{hyp \vdash g \rightarrow f}$$

Anteceding step1 is, however, merely preparatory to steps4 and 5, which accomplish the overall goal of generalizing the conclusion of step1. The pattern of steps 2 through 5 therefore define a second derived inference rule, *generalization.*

generalize (x, t)
where t is a theorem of the form hyp $\vdash$ f

$$\frac{\text{hyp} \vdash f}{\text{hyp} \vdash \forall x.\ f}$$

Using the new rule of generalization, the proof of reflex$\equiv$ is shortened and made considerably more conspicuous.

4.4. trans→

step1: Assume {(f → g), (g → h)} (g → h)
step2: antecede(f, step1)
step3: S {} (f, g, h)
step4: MP(step3, step2)
step5: Invoke(step4, (f → g))
step6: MP(step4, step5)
step7: Discharge(step6, (g → h))
trans→: Discharge(step7, (f → g))

The conclusion of a theorem is often less interesting than the method of its proof. If we recall our motivating analogy between proving and programming this is hardly surprising; a procedure is often more interesting and valuable than any of its instances (applications). This is the case with trans→, which leads to a useful inference rule, *trans*, but which is almost never cited *qua* theorem. Steps 2 through 6 supply the pattern of inferences for the rule.

trans (t1, t2)
where t1 is a theorem of the form hyp $\vdash$ (f $\rightarrow$ g)
and t2 is a theorem of the form hyp $\vdash$ (g $\rightarrow$ h)

$$\frac{\text{hyp} \vdash f \rightarrow g \qquad \text{hyp} \vdash g \rightarrow h}{\text{hyp} \vdash f \rightarrow h}$$

4.5. descrip

Henceforth, we will introduce new inference rules by indenting the sequence of proof steps that they abstract immediately after their first use.

step1: description {} (z, y, (g $\leftrightarrow$ y()))
step2: comprehension {} (y, g)
step3: apply(step1, step2)
step3a: conjoin(step1, step2)
step3b: instantiation {} (z, (z$\equiv$!y.(g $\leftrightarrow$ y()) $\leftrightarrow$ $\forall$y.((g $\leftrightarrow$ y()) $\leftrightarrow$ y$\equiv$z)))
step3c: Sub(!y.(g $\leftrightarrow$ y()), z, step3b)
step3d: MP(step3c, step3a)
step4: simp1 step3
step4a: p1 {} (!y(g $\leftrightarrow$ y())$\equiv$!y(g $\leftrightarrow$ y()) $\rightarrow$ $\forall$y((g $\leftrightarrow$ y()) $\leftrightarrow$ y$\equiv$!y(g $\leftrightarrow$ y())),
$\forall$y((g $\leftrightarrow$ y()) $\leftrightarrow$ y$\equiv$!y(g$\leftrightarrow$ y())) $\rightarrow$!y(g $\leftrightarrow$ y())$\equiv$!y(g $\leftrightarrow$ y()))
step4b: MP(step4a, step3)
step5: Sub(!y(g $\leftrightarrow$ y()), y, reflex$\equiv$)
step6: MP(step4, step5)
step7: apply(step6, step2)
step8: simp2 step7
step8a: p2 {} ((g $\leftrightarrow$ (!y.g $\leftrightarrow$ y())()) $\rightarrow$!y(g $\leftrightarrow$ y())$\equiv$!y(g $\leftrightarrow$ y()),
!y(g $\leftrightarrow$ y())$\equiv$!y(g $\leftrightarrow$ y()) $\rightarrow$ (g $\leftrightarrow$ (!y.g $\leftrightarrow$ y())()))
step8b: MP(step8a, step7)
descrip: MP(step8, step5)

Once the new inference rules have been defined and their defining patterns eliminated, the remaining nine steps of the proof are quite compact and natural, even though they represent a fairly lengthy "formal derivation" in the sense of [3].

4.6. in1

step1: Assume {(f ∧ **E**z), ((f → z()) ∧ (g → z()))} (f ∧ **E**z)
step2: simp1 step1
step3: Invoke(step2, ((f → z()) ∧ (g → z())))
step4: simp1 step3
step5: MP(step4, step2)
step6: Discharge(step5, ((f → z()) ∧ (g → z())))
step7: Discharge(step6, (f ∧ **E**z))
in1: Curry step7

4.7. in2

step1: Assume {(g ∧ **E**z), ((f → z()) ∧ (g → z()))} (g ∧ **E**z)
step2: simp1 step1
step3: Invoke(step2, ((f → z()) ∧ (g → z())))
step4: simp2 step3
step5: MP(step4, step2)
step6: Discharge(step5, ((f → z()) ∧ (g → z())))
step7: Discharge(step6, (g ∧ **E**z))
in1: Curry step7

Even though we have not discussed the relationship between proofs and programs, it is instructive to compare the programs that **Intuit** produces from the proofs of in1 and in2.

in1: λx.λy.λz.(fst z)(x)
in2: λx.λy.λz.(snd z)(x)

These programs act as injections into the first and second summands, respectively, of a binary disjoint union (∨!). Their similarity of structure is a direct consequence of the similarity of the corresponding proofs. The only difference is the use of the first or second projection function, corresponding to the use of simp1 or simp2 at step 4 in the proof.

4.8. copair

The notions of disjunction, negation, existential quantification, falsehood, and truuth are not primitives in **Intuit**. Instead, they are composed from the primitive

conneectives according to the definitions given at the beginning of **§3**. The theorem copair, together with in1 and in2, establishes that the definition of $\vee$ behaves as it should. Problems 11-15 similarly establish the key properties of the other defined notions.

Let H $\triangleq$ {(f → g), (h → g), (f$\vee$h)}.

step1: Assume H (f$\vee$h)
step2: comprehension H (y, g)
step3: apply(step1, step2)
step4: AddHyp(H, descrip)
step5: simp1 step4
step6: Invoke(step5, (f → g))
step7: trans(step6, step5)
step8: Invoke(step7, (h → g))
step9: trans(step8, step5)
step10: conjoin(step7, step9)
step11: MP(step3, step10)
step12: conjoin(step4, step11)
step13: Discharge(step12, (f$\vee$h))
step14: Discharge(step13, (h → g))
copair: Discharge(step14, (f → g))

4.9. strengthen

step1: Assume {(f → h), (g$\wedge$f)} (f → h)
step2: Invoke(step1, (g$\wedge$f))
step3: simp2 step2
step4: MP(step1, step3)
step5: Discharge(step4, (g$\wedge$f))
strengthen: Discharge(step5, (f → h))

After the earlier exercises, this one should have been a snap; I placed it here as a confidence-builder.

4.10. sym≡

Let H $\triangleq$ {x≡y}.

step1: Assume H (x≡y)
step2: antecede(x≡x, step1)
step3: Sub(x, y, reflex≡)
step4: AddHyp(H, step3)
step5: antecede(x≡y, step4)
step6: conjoin(step5, step2)
step7: generalize(x, step6)
step8: extensionality H (x, y, x)
step9: MP(step8, step7)
sym≡: Discharge(step9, x≡y)

4.11. absurdity

Let Ha $\triangleq$ {(f → g), (f → ¬g)}, and Hb $\triangleq$ {(**E**z), f}.

step1: Assume Hb (**E**z)
step2: Invoke(step1, (f → ¬g))
step3: Invoke(step2, f)
step4: MP(step2, step3)
step5: apply(step4, step1)
step6: Invoke(step5, (f → g))
step7: MP(step6, step3)
step8: MP(step5, step7)
step9: Discharge(step8, f)
step10: Discharge(step9, (**E**z))
step11: elaborate([**E**z; f→f; f→z()], [f; g; h], strengthen)
step12: AddHyp(Ha, step11)
step13: MP(step12, step10)
step14: Curry step13
step15: AddHyp(Ha, id)
step16: MP(step14, step15)
step17: Discharge(step16, (f → ¬g))
absurdity: Discharge(step17, (f → g))

4.12. contradiction

Let H $\triangleq$ {f, ¬f}.

step1: Assume H (¬f)
step2: comprehension H (z, g)
step3: apply(step1, step2)
step4: Invoke(step3, f)
step5: MP(step3, step4)
step6: AddHyp(H, descrip)
step7: simp2 step6
step8: MP(step7, step5)
step9: Discharge(step8, f)
contradiction: Discharge(step9, ¬f)

4.13. equant

step1: Assume {((f∧(**E**x))∧(**E**z)), (∀x. f → z())} ((f∧(**E**x))∧(**E**z))
step2: simp2 step1
step3: simp1 step1
step4: simp1 step3
step5: step2 step3
step6: Invoke(step5, (∀x. f → z()))
step7: apply(step6, step5)
step8: MP(step7, step4)
step9: Discharge(step8, (∀x. f → z()))
step10: Discharge(step9, ((f∧(**E**x))∧(**E**z)))
equant: Curry step10

4.14. initial⊥

step1: Assume {⊥} (⊥)
step2: comprehension {⊥} (y, f)
step3: apply(step1, step2)
step4: elaborate([f], [g], descrip)
step5: AddHyp({⊥}, step4)
step6: simp2 step5
step7: MP(step6, step3)
initial⊥: Discharge(step7, ⊥)

4.15. true

Let kf $\triangleq$ f → (g → f), and ky $\triangleq$!y.kf ↔ y(). Let Ha $\triangleq$ {x≡ky}, Hb $\triangleq$ {x()}.

step1: comprehension {} (y, kf)
step2: description {} (y, x, x())
step3: apply(step2, step1)
step4: Assume Ha (x≡ky)
step5: Sub(ky, y, symequiv)
step6: AddHyp(Ha, step5)
step7: MP(step6, step4)
step8: elaborate([kf], [g], descrip)
step9: simp1 step8
step10: K {} (f, g)
step11: MP(step9, step10)
step12: AddHyp(Ha, step11)
step13: conjoin(step12, step7)
step14: substitutivity Ha (x(), x, ky, x)
step15: MP(step14, step13)
step16: Discharge(step15, (x≡ky))
step17: Assume Hb (x())
step18: generalize(x, step17)
step19: AddHyp(Hb, initiality)
step20: MP(step19, step18)
step21: Discharge(step20, x())
step22: elaborate([x≡ky], [f], step21)
step23: conjoin(step22, step16)
step24: generalize(x, step23)
step25: simp2 step3
step26: MP(step25, step24)
step27: conjoin(step1, step26)
step28: substitutivity newenv (**E**z, z, ky, !x.x())
true: MP(step28, step27)

The reader who is familiar with the semantics of **Intuit** will be gratified to know that the program resulting from this proof is simply *nil*, the simplest semantics possible for truth in the model.

Acknowledgements

Eight of the exercises, namely nos. 5-7 and 10-14, are due to Fourman [4]. The others were developed specifically to help in solving those eight.

References

1. J. L. Bates and R. L. Constable, Proofs as Programs, *ACM Trans. Prog. Lang. and Systems 7*, 1 (January 1985), 113-136.

2. R. L. Constable and D. R. Zlatin, The Type Theory of PL/CV3, *ACM Trans. Prog. Lang. and Systems 6*, 1 (January 1984), 94-117.

3. R. A. DeMillo, R. J. Lipton and A. J. Perlis, Social Processes and Proofs of Theorems and Programs, *Comm. ACM 22*, 5 (May 1979), 271-280.

4. M. P. Fourman, The Logic of Topoi, in *Handbook of Mathematical Logic*, J. Barwise (ed.), North-Holland, 1977, 1054-1090.

5. M. J. Gordon, A. J. Milner and C. P. Wadsworth, Edinburgh LCF, in *Lecture Notes in Computer Science, Vol. 78*, Springer, Berlin, 1979.

6. P. Martin-Löf, Constructive Mathematics and Computer Programming, *6th Int'l. Congress for Logic, Methodology, and Philosophy of Science*, Hannover, Aug. 1979.

7. A. Meyer, The Inherent Computational Complexity of Theories of Ordered Sets: a Brief Survey, in *Int. Congress of Mathematicians*, Aug. 1974.

8. B. Nordström, Programming in Constructive Set Theory: Some Examples, *Proc. 1981 ACM Conf. on Functional Prog. Lang. and Comp. Architecture*, Portsmouth, NH, Oct. 1981, 141-153.

9. J. Shultis, Deduction and Abstraction in Intuit, An Intuitionistic Programming System, University of Colorado Dept. of Computer Science Technical Report, forthcoming, 1985.

10. J. Shultis, A Realizability Semantics for the Logic of Topoi, University of Colorado Dept. of Computer Science Technical Report, forthcoming, 1985.

11. L. Stockmeyer, The Complexity of Decision Problems in Automata Theory and Logic, Ph.D. Thesis, MIT, 1974.