Mechanizing the Axiomatic Semantics for a Programming Language with Asynchronous Send and Receive in HOL. [1]

William L. Harrison
Division of Computer Science
University of California, Davis

June, 1992

[1]This work was supported by the National Security Agency under Agreement DOD-MDA 904-91-C-7053, and was supervised by Sami Sajdjari.

Abstract

This thesis presents the axiomatic semantics for a simple distributed language and its mechanization in HOL. The constructs of this language include asynchronous **send** and synchronous **receive** statements as well as those basic to a sequential programming language. The language has the appearance of a system programming language that supports sequential execution extended with message passing, and would be a suitable basis for coding distributed operating systems. Included in the mechanization are functions which generate the goals associated with the verification of sequences of simple statements in the language. In contrast to Gordon's (and others) work on the HOL mechanization of programming logics, the starting point for our work is not a denotational definition of the language. Due to the presense of many *possible* global states in an arbitrary distributed program, it is not clear that a denotational definition of a distributed language would yield a tractable basis for specification and verification of parallel programs. We chose an *axiomatic* approach to the specification of the target language, rather than the constructive, denotational approach. That is, we associate *Hoare triples* with each programming language construct instead of a function as with the denotational style. In addition to presenting examples of distributed programs verified using our mechanization, we present preliminary thoughts towards a verified secure distributed system, the security kernel of which is written in an extension of this simple distributed language.

Contents

List of Figures

Acknowledgements

This work was supported by the National Security Agency's University Research Program, under contract DOD-MDA 904-91-C-7053. Sami Sajdjari provided us with technical direction relating to the properties and features of secure distributed systems of interest to the security community.

Chapter 1

Introduction

As a rule, the more complex a system design becomes, the more likely it contains errors. These errors may be the result of carelessness on the part of the designer(s), or from the lack of a clear understanding of the design problem. The Computer Science discipline known sometimes as "formal methods" attempts to combat these errors through the *formal specification and verification* of system designs, which entails giving a rigorous mathematical meaning or model of the design problem (specification), and demonstrating that this model has certain desirable properties (verification). A specification of a programming language is usually referred to as a "semantics" for the language. Verification of systems is frequently performed with an automated theorem prover because doing so provides greater assurance of proof correctness than with a "proof by hand".

Writing programs is a notoriously error-prone task, but writing concurrent programs is especially error-prone due to the non-deterministic quality of asynchronously-executing intercommunicating code. Consequently, specification and verification of concurrent programs is particularly desirable because it forces the programmer/designer to consider whether the code actually fulfills the designer's expectations.

Traditionally, the HOL community has preferred *definitional* or *constructive* approaches to specification as opposed to axiomatic ones. Defining a new term in an existing theory can not introduce inconsistency to that theory, whereas a new ill-conceived axiom can. The

preference for *definitional extensions* to an existing specification or theory is, therefore, based on the desire to avoid inconsistency. However after completing this project, we believe that a specification for a distributed language involves complexities which would make a definitional approach intractable. Apparently, the HOL community agrees with us, if a perusal of the last few *HOL Users Group Conference Proceedings* is any indication. Our approach is novel in the context of the HOL community in that it is brazenly and unabashedly axiomatic. Ours is the first mechanization of a parallel language in HOL. While it is certainly important to know that there *is* a language which is a model of the axioms, it is also important that specification and verification of complex systems *actually* happens as well.

In his paper "Mechanizing Programming Logics in Higher Order Logic"[15], Michael Gordon describes his mechanization into HOL of the Hoare semantics for a simple imperative programming language. From the denotational semantics of his language, Gordon's mechanization introduces a sentence defining a Hoare triple for each statement; the starting-point interpretation of a programming language statement is its *denotational meaning* — a function on the state of the program which indicates the new state resulting from the execution of the statement. This is in contrast to most approaches to the mechanization of programming logics, where the starting point *is* a Hoare-triple semantics. Gordon's approach is purely *definitional*; that is, it does not introduce axioms (Hoare triples) and, hence, will not introduce inconsistencies into the system.

Our work had as its goal the mechanization of the semantics of a distributed programming language, one in which asynchronously executing processes communicate with each other by *send*ing and *receive*ing messages; communicating processes could be on the same machine or on different machines. We envision this language as being the basis for a simple low-level system language, suitable for writing distributed operating systems and applications. We intended to follow Gordon's definitional approach, but immediately ran into what seems to be an insurmountable problem: the difficulty of defining a global state for an arbitrary distributed program when communication is asynchronous. Hence, we retreated from the definitional approach, and decided instead on an axiomatic approach. Our starting point, then, was the axiomatic definition of message passing developed by Schlichting [24]. In

our language, *send* is asynchronous—a processing sending a message to another process does not block. On the other hand, *receive* is synchronous—a receiving process blocks if no appropriate message is pending. For the sequential features of the language, we settled on the RISC-inspired assembly language for the AVM-1 microprocessor verified by Windley [28].

Our mechanization is in terms of axiom schemata: an ML function is provided for each statement type which creates an axiom instance appropriate to the particular components of the statement. Our mechanization package includes help functions which assist the user in the application of inference rules to complicated Hoare formulas. The method used is sufficiently flexible to easily mechanize other programming language semantics.

Chapter 2 presents a brief review of the Hoare triple approach to the definition of sequential programming languages and a description of Gordon's work. Chapter 3 briefly introduces[1] the HOL theorem proving system. Chapter 4 presents the rules developed by Schlichting for *send* and *receive* without their derivation. In chapter 5, the distributed target language L we are defining is described, and the programming logic $\mathcal{PL}$ defining its axiomatic semantics is introduced. Chapter 6 describes the HOL mechanization of the axiomatic semantics of the target language. Two annotated example derivations in the system are presented in chapter 7. A discussion of our work and plans for extensions and verified applications of the target language are presented in chapter 8. In particular, we are working towards a verified "distributed stack", inspired by the CLI verified stack [3, 4]. The "UCD tower" will ideally be a distributed version of the CLI stack; that is, it will eventually include verified hardware and software components such as a verified multiprocessor, distributed operating system kernel, assembler, and compiler. The appendices contain listings of all the ML code for our mechanization.

[1] We thank Phillip Windley for allowing us to include material from his doctoral dissertation[28] in this chapter (specifically, sections 3.1-3.2).

Chapter 2

Axiomatic Semantics of Sequential Programming Languages

An axiomatic (or "Hoare") semantics codifies the semantics of a programming language into a formal system or calculus with axioms and inference rules. In a Hoare-style calculus [16], the well-formed formulas are triples:

$$\{\phi\}\ \ StatementList\ \ \{\psi\}$$

where ϕ and ψ are formulas in some (usually first-order) language, and *StatementList* is a list of commands in the programming language being specified. ϕ and ψ are the *precondition* and *postcondition*, respectively, of the above triple. The intended interpretation of such a triple is, if ϕ is *true of the system state* before sequential execution of the commands in *StatementList*, then ψ is true of the system state after termination of *StatementList*. A Hoare triple specifies the *partial* correctness of a statement list—termination must generally be demonstrated by some other means. *Total* correctness consists of partial correctness and termination. Generally, it is taken for granted that "true of the system" is well understood and well-defined [1].

An example axiom schema for an imperative, sequential programming language is the assignment axiom schema:

$$\{\ \phi^{x}_{expr}\ \}\ \ x\ :=\ expr\ \ \{\ \phi\ \}$$

That is, if ϕ^x_{expr} is true [1] before execution of "$x := expr$", then ϕ is true after execution. A typical inference rule is:

$$\frac{\{\ \phi\ \}\ \ StatementList\ \ \{\ \psi\ \},\quad \psi \implies \theta}{\{\ \phi\ \}\ \ StatementList\ \ \{\ \theta\ \}}$$

That is, if $\psi \implies \theta$ is a theorem of some (generally first-order) theory, and

$$\{\ \phi\ \}\ \ StatementList\ \ \{\ \psi\ \}$$

is a theorem of our programming logic, then

$$\{\ \phi\ \}\ \ StatementList\ \ \{\ \theta\ \}$$

is also a theorem of our programming logic. This rule is also known as *postcondition weakening.*

It should be observed that the above axiom and rule are entirely *syntactic* in nature. That is, they are defined completely in terms of the properties of strings. Axioms and inference rules of Hoare-style semantics have this quality. A *proof* of a formula ϕ in a formal system can be defined as a sequence of well-formed formulas (triples in the case of Hoare semantics):

$$\phi_0,\ \phi_1,\ \phi_2, \ldots$$

where each ϕ_i is either an axiom, or follows from $\phi_0, \ldots, \phi_{i-1}$ by some rule of inference, and ϕ_n is ϕ. It should be noted that if the axioms and inference rules of a formal system are syntactically defined, then a proof is also a syntactic object. That is, a proof of a theorem in the system depends only upon syntactic rules, rather than on the intended interpretation of the symbols. In such a formal system, the notions *proof* or *derivation* are entirely divorced from their intended meaning. This is a desirable property for a formal calculus because proofs in such a logic can be readily checked mechanically by a theorem proving system such as HOL.

[1] ϕ^x_{expr} refers to the formula obtain by substituting $expr$ for all free occurrences of x (also known as the *simultaneous substitution* of $expr$ for x).

Expressions:

$$\mathcal{E} \;::=\; \mathcal{N} \mid \mathcal{V} \mid \mathcal{E}_1 + \mathcal{E}_2 \mid \mathcal{E}_1 - \mathcal{E}_2 \mid \mathcal{E}_1 \times \mathcal{E}_2 \;\ldots$$

Booleans:

$$\mathcal{B} \;::=\; \mathcal{E}_1 = \mathcal{E}_2 \mid \mathcal{E}_1 \geq \mathcal{E}_2 \;\ldots$$

Commands:

$$\begin{aligned}\mathcal{C} \;::=\;& \textbf{skip}\\ &\mid \mathcal{V} := \mathcal{E}\\ &\mid \mathcal{C}_1; \mathcal{C}_2\\ &\mid \textbf{if } \mathcal{B} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2\\ &\mid \textbf{while } \mathcal{B} \textbf{ do } \mathcal{C}\end{aligned}$$

Figure 2.1: Backus-Naur Description of Gordon's Language

2.1 Previous Work

In his paper "Mechanizing Programming Logics in Higher Order Logic"[15], Michael Gordon describes his mechanization into HOL of the Hoare semantics for a simple imperative programming language. A Backus-Naur description of this language is given in Figure 2.1. There, the variable $\mathcal{N}$ ranges over the *numerals* $\underline{0}$, $\underline{1}$, $\underline{2}$, ..., the variable $\mathcal{V}$ ranges over *program variables* X, Y, Z, ..., the variable $\mathcal{B}$ ranges over *boolean expressions*, the variables $\mathcal{E}$, $\mathcal{E}_1$, and $\mathcal{E}_2$ range over *integer expressions*, and the variables $\mathcal{C}$, $\mathcal{C}_1$, and $\mathcal{C}_2$ range over *commands*.

As an example of how the axiomatic semantics of this language are mechanized in HOL, the case for the assignment statement "$\mathcal{V} := \mathcal{E}$" will be explained. Gordon's mechanization uses a relational form of the well-known denotational semantics of this language to derive the language's axiomatic semantics. The details of how this is accomplished for the assignment statement follow.

The denotational semantics for the assignment statement [26] are usually given by:

$$[\![\ v\ :=\ e\ ]\!]\ =\ (\lambda s : \mathbf{State}.\ \lambda x : \mathbf{Num}.\ (\ x\ =\ v\ \longrightarrow\ (e\ s)\ |\ s\ ) \tag{2.1}$$

The assignment "$v := e$" denotes the function $[\![\ v\ :=\ e\ ]\!]$ with type signature "$: \mathbf{State} \rightarrow (\mathbf{Num} \rightarrow \mathbf{State})$". Given a state s and a program variable $x : \mathbf{Num}$ (we are assuming all program variables have type **Num** in this little language), the function $[\![\ x := expr\ ]\!]$ returns a state identical to s except that x is bound to the value $(expr\ s)$, where $(expr\ s)$ is the value of expression $expr$ in state s. $[\![\ x := expr\ ]\!]$ is referred to as the *denotational meaning* of the statement "$x := expr$".

The axiom schema for the assignment command [1, 16, 21] is:

$$\{\ \phi_e^v\ \}\ \ v\ :=\ e\ \ \{\ \phi\ \}.$$

as asserted in the introduction. Gordon's method converts the above functional definition from the previous paragraph into a *relational* one that can be readily translated into HOL. The relational form of (2.1)—Assign—is defined as:

$$\mathsf{Assign}(v, e)(s_1, s_2)\ =\ (s_2\ =\ \mathsf{Bnd}(e, v, s_1))$$

where:

$$\mathsf{Bnd}(e, v, s)\ =\ \lambda x.\ (x = v\ \longrightarrow\ (e\ s)|s).$$

$\mathsf{Assign}(v, e)$ is a relation of type "**State**×**State**" which is true for a **State** pair if and only if s_2 is the **State** which results from binding the variable v to the value of expression e in state s_1. Note furthermore that Bnd is shorthand for equation (2.1).

One further definition is necessary to complete the definition of the mechanization of the assignment statement. Gordon interprets a Hoare triple $\{\ \phi\ \}\ \ Com\ \ \{\ \psi\ \}$ as meaning:

$$\forall s_1\ s_2 : \mathbf{State}.\ \ [\![\ \phi\ ]\!]\ s_1\ \wedge\ [\![\ Com\ ]\!]\ (s_1, s_2)\ \ \Longrightarrow\ \ [\![\ \psi\ ]\!]\ s_2$$

where $[\![\ Com\ ]\!]\ (s_1, s_2)$ is a relation which is true if and only if state s_2 results from the execution of the command Com in state s_1. This sentence specifies the partial correctness of the command Com with respect to the pre- and postconditions ϕ and ψ. That is, if ϕ is true is state s_1 and command $[\![\ Com\ ]\!]\ (s_1, s_2)$ is true, then ψ is true in state s_2. If

the command Com does *not* terminate when started in state s_1, then $[\![\ Com\ ]\!]\,(s_1, s_2)$ is false, and so the entire partial correctness relation is true. The above formula is abbreviated as: $\mathsf{Spec}\,(\phi,\ Com,\ \psi)$, and specifies the partial correctness of Com, which is the intended interpretation of a triple.

One can pretty-print a triple $\{X = 1\}\ \ X := X + 1\ \ \{X = 2\}$ using Spec and Assign in HOL by:

$$\mathsf{Spec}\,(\,[\![\,X = 1\,]\!]\,),\ \mathsf{Assign}\,('X',\ [\![\,X + 1\,]\!]\,),\ [\![\,X = 2\,]\!]\,).$$

Hoare triples in this mechanization are represented by abbreviations of a relational form of the target language's denotational semantics.

In this setting, the axiom schemata and inference rules of a Hoare semantics are HOL theorems—Gordon derives them from the definitions of Assign, Spec, Bnd, etc. The assignment axiom schema for "$v := expr$" is:

$$\mathsf{Spec}\,(\,[\![\,\phi[expr/v]\,]\!]\,,\mathsf{Assign}('v',\ [\![\,expr\,]\!]\,),\ [\![\,\phi\,]\!]\,)$$

for arbitrary v and $expr$. The universal closure of the above formula is then derived, thus demonstrating that the representation of the axiom schema for assignment holds for any variable v and expression $expr$ in the logic.

There are many advantages to mechanizing a programming logic in this manner. For one, since the axiomatic semantics are *derived* from the denotational, there must be a programming language which fulfills the axiomatic semantics. That is, a *model* exists of the syntactically defined rules and axioms. The axioms have meaning, in other words. This approach is constructive and definitional in that sense, and is quite elegant for these reasons.

However, difficulties arise with this approach when other more complicated programming language constructs are considered. The language mechanized in this thesis has asynchronous **send** and **receive**, the denotational semantics of which are (to the best of my knowledge) not well understood. The difficulty in defining a denotational semantics for asynchronous programming language constructs stems from the fact that generally there are many *possible* states arising from the execution of a concurrent statement, rather than a single resulting state as with sequential statements. Denotational meanings are given to

asynchronous constructs through *powerdomain construction* (see [25]). Gordon's method can not even be *attempted* until these denotational semantics are well understood. Because his target language is limited to a very simple, imperative sequential language, the difficulties associated with semantically more complicated programming language constructs are completely side-stepped.

Chapter 3

HOL Overview

HOL is a general theorem proving system developed at the University of Cambridge [6, 14] that is based on Church's theory of simple types, or higher order logic [7]. Church developed higher order logic as a foundation for mathematics, but it can be used for describing and reasoning about computational systems of all kinds. Higher order logic is similar to the more familiar predicate logic, but allows quantification over predicates and functions, not just variables, allowing more general systems to be described.

HOL grew out of Robin Milner's LCF theorem prover [13] and is similar to other LCF progeny such as NUPRL [10]. Because HOL is the theorem proving environment used in the body of this work, we will describe it in more detail.

HOL's proof style can be tailored to the individual user, but most users find it convenient to work in a goal-directed fashion. HOL is a tactic based theorem prover. A tactic breaks a goal into one or more subgoals and provides a justification for the goal reduction in the form of an inference rule. Tactics perform tasks such as induction, rewriting, and case analysis. At the same time, HOL allows forward inference and many proofs are a combination of both forward and backward proof styles. Any theorem proving strategy a user employs in connection with HOL is checked for soundness, eliminating the possibility of incorrect proofs.

HOL provides a metalanguage, ML, for programming and extending the theorem prover.

Using ML, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be combined into new theories for later use. The metalanguage makes the HOL verification system extremely flexible.

In HOL, all proofs, even tactic-based proofs, are eventually reduced to the application of inference rules. Most nontrivial proofs require large numbers of inferences. Proofs of large devices such as microprocessors can take many millions of inference steps. In a proof containing millions of steps, what kind of confidence do we have that the proof is correct? One of the most important features of HOL is that it is *secure*, meaning that new theorems can only be created in a controlled manner. HOL is based on five primitive axioms and eight primitive inference rules. All high-level inference rules and tactics do their work through some combination of the primitive inference rules. Because the entire proof can be reduced to one using only eight primitive inference rules and five primitive axioms, an independent proof-checking program could check the proof syntactically.

3.1 The Language

The HOL language is described in this section. We will discuss HOL's terms and types.

3.1.1 Terms

All HOL expressions are made up of terms, of which there are four kinds—variables, constants, function applications, and abstractions (lambda expressions). Variables and constants are denoted by any sequence of letters, digits, underlines, and primes starting with a letter. Constants are distinguished in the logic; any identifier that is not a distinguished constant is taken to be a variable. Constants and variables can have any finite arity, not just 0, and, thus, can represent functions as well.

Function application is denoted by juxtaposition, resulting in a prefix syntax. Thus, a term of the form `"t1 t2"` is an application of the operator `t1` to the operand `t2`. The term's value is the result of applying `t1` to `t2`.

Table 3.1: HOL Infix Operators

Operator	*Application*	*Meaning*
=	t1 = t2	t1 equals t2
,	t1,t2	the pair t1 and t2
$\wedge$	t1 $\wedge$ t2	t1 and t2
$\vee$	t1 $\vee$ t2	t1 or t2
$\Longrightarrow$	t1 $\Longrightarrow$ t2	t1 implies t2

Table 3.2: HOL Binders

Binder	*Application*	*Meaning*
$\forall$	$\forall$ x. t	for all x, t
$\exists$	$\exists$ x. t	there exists an x such that t
ε	ε x. t	choose an x such that t is true

An abstraction denotes a function and has the form "λ x. t". Its two parts are: the bound variable x and the body of the abstraction t. An abstraction represents a function, f, such that "f(x) = t". For example, "λ y. 2*y" denotes a function on numbers which doubles its argument.

Constants can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written "rand1 op rand2" instead of in the usual prefix form: "op rand1 rand2". Table 3.1 shows several of HOL's built-in infix operators.

Constants can also belong to another special class called binders. A familiar example of a binder is $\forall$. If c is a binder, then the term "c x.t" (where x is a variable) is written as shorthand for the term "c(λ x. t)". Table 3.2 shows several of HOL's built-in binders.

In addition to the infix constants and binders, HOL has a conditional statement that is written a $\rightarrow$ b | c, meaning "if a, then b, else c."

3.1.2 Types.

HOL is strongly typed to avoid Russell's paradox and others like it. Russell's paradox occurs in a high order logic when one can define a predicate that leads to a contradiction. Specifically, suppose that we define P as P(x) = $\neg$x(x) where $\neg$ denotes negation. P is true when its argument applied to itself is false. Applying P to itself leads to a contradiction

Table 3.3: HOL Type Operators

Operator	*Arity*	*Meaning*
`bool`	0	booleans
`ind`	0	individuals
`num`	0	natural numbers
`(*)list`	1	lists of type `*`
`(*,**)prod`	2	products of `*` and `**`
`(*,**)sum`	2	coproducts of `*` and `**`
`(*,**)fun`	2	functions from `*` to `**`

since `P(P) = ¬P(P)` (i.e., $true = false$). This kind of paradox can be prevented by typing since, in a typed system, the type of `P` would never allow it to be applied to itself.

Every term in HOL is typed according to the following recursive rules:

1. Each constant or variable has a fixed type.
2. If `x` has type α and `t` has type β, the abstraction λ `x. t` has the type $(\alpha \rightarrow \beta)$.
3. If `t` has the type $(\alpha \rightarrow \beta)$ and `u` has the type α, the application `t u` has the type β.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (`*`) followed by a (possibly empty) sequence of letters and digits. Thus, `*`, `***`, and `*ab2` are all valid type variables. All type variables are implicitly universally quantified, yielding type polymorphic expressions.

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\sigma_1, \ldots, \sigma_n$ are types and `op` is a type operator of arity n, then $(\sigma_1, \ldots, \sigma_n)$`op` is a type. Note that type operators are postfix while normal function application is prefix or infix. A type operator of arity 0 is a type constant.

HOL has several built-in types, which are listed in Table 3.3. The type operators `bool`, `ind`, and `fun` are primitive. HOL has a special syntax that allows `(*,**)prod` to be written as `(* # **)`, `(*,**)sum` to be written as `(* + **)`, and `(*,**)fun` to be written as `(* -> **)`.

3.2 The Proof System

HOL is not an automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

1. Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories contain the five axioms that form the basis of higher order logic as well as a large number of theorems that follow from them.

2. Rules of inference for higher order logic. These rules contain not only the eight basic rules of inference from higher order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

3. A collection of tactics. Examples of tactics include: `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition; `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms; and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply each other.

4. A proof management system that keeps track of the state of an interactive proof session.

5. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

3.3 The Inductive Relation Definitions Package

We include in this section a sample application of the inductive relation definitions package of HOL. The particular example we use is borrowed from [20]. We employ the inductive

```
#let (rules,ind) =

% typing %  let Even = "Even:num->bool" in
              new_inductive_definition
% infix∃ %     false
% name %       `Even`
% pattern %    ("^Even n", [])
% rule 1 %   [[
              %------------------------------%],
                            "^Even 0"              ;

% rule 2 %   [               "^Even n"
              %------------------------------%],
                            "^Even (n+2)"           ];;
```

Figure 3.1: Definition of `Even`

relation definitions package to define the inference rules of our programming logic in Chapter 6. While this application is larger than the simple example presented here (i.e., more rules), it is not any more complicated.

The relation "is an even natural number" can be defined inductively on the natural numbers (i.e., 0, 1, 2,...) with the following:

- 0 is an even natural number, and
- if n is an even natural number, then so is $n+2$.

The two inductive rules above can be readily formalized in HOL using the HOL function `new_inductive_definition` as is shown in Figure 3.1.

An application of `new_inductive_definition` takes an boolean infix flag, a name string for the relation, a pattern, and a list of inductive rules. These arguments are labeled appropriately in Figure 3.1, wherein a relation "`Even`" is being defined. The infix field specifies whether the defined relation will be written with infix style (e.g, "$<$ 0 1" rather than "$0 < 1$"). Here, we indicate that `Even` is not written with infix style with the parameter "false". The string name of the relation to be defined is `Even`. The third argument is a "pattern" which supplies information to the HOL interpreter necessary to define a *class*

```
rules =
     [ Even 0;
       ∀ n. Even n ⟹  Even (n+2) ] : thm list

ind =
     ⊢ (∀ P. P 0 and (∀ n. P n ⟹  P (n+2))) ⟹
               ∀ n. Even n ⟹  P (n+2)
```

Figure 3.2: Output of `new_inductive_definition`

of inductive relations. Here, however, we define a *single* relation, rather than a class of relations. We refer the interested reader to [20] for a detailed description of the pattern field, as a complete description of this field would be considerably beyond the scope of this presentation. Only single constant relations are defined in this thesis. The next field is a list of rules. Each rule consists of premises, which occur above the dotted line in each rule, and conclusions, which occur below the dotted line in Figure 3.1. Rule 1 in Figure 3.1 has no premises, but has the conclusion "`^Even 0`". This corresponds to the informal base case that "0 is an even natural number". The antiquotation operator `^` *coerces* an ML variable into an HOL variable but has no logical meaning. Because `Even` is defined by a `let` clause, it is an ML variable rather than an HOL variable. Rule 2 in Figure 3.1 has one premise "`^Even n`", and one conclusion "`^Even (n+2)`". The corresponding informal rule is "if n is an even natural number, then so is *(n+2)*."

Applying `new_inductive_definition` automatically proves the existence of the least predicate closed under the set of rules, and returns a pair consisting of a theorem list and a rule induction theorem as shown in Figure 3.2. The theorem list contains theorems which state that the inductive rules hold of the newly defined relation. In Figure 3.2, the ML identifier `rules` is assigned the theorem list corresponding to rules 1 and 2 in Figure 3.1. The rule induction theorem returned by the above application of `new_inductive_definition` states that if an arbitrary relation `P` is true on the same set of naturals as `Even` (i.e., the even naturals), then `Even n` implies `P (n+2)`.

Chapter 4

Axiomatic Semantics of Asynchronous Message Passing

Transmission of messages or datagrams across a communication network can not ensure that the delivery will be reliable. Data may be sent and not delivered, so messages may not arrive in the order sent. Messages may also be discarded due to noise corruption or because message buffers are already full at the destination site.

Typically, network software layers are employed to compensate for these difficulties [27]. Such network software creates a facility ensuring reliable message transmission. As part of his doctoral dissertation at Cornell University, Rick Schlichting developed Hoare-style proof rules for asynchronous **send** and **receive** commands which may be implemented by typical network software.

4.1 The Format of send and receive

The **send** statement has the following format:

send *expr* **to** *dest*,

and is executed as follows:

1. the value of *expr* is evaluated,

2. a message with value *expr* is sent to the process named *dest*.

Execution of **send** is *asynchronous*, meaning that the sending process continues execution while the message is transmitted. This is in contrast to the *synchronous* output statement "*dest*!*message*" of CSP [17] which blocks the execution of the sender process until the message is received. We do not handle the case where *dest* is an invalid process name, nor do we make any assumptions at this point about *how* the processes are named. We assume only that for each reference to *dest*, that *dest* is an expression which evaluates to the name of an existing process.

The **receive** statement has the form:

$$\textbf{receive } m \textbf{ when } \beta,$$

where m is a program variable and β is a (usually first-order) expression involving m and other program variables.

Execution of a **receive** statement proceeds as follows:

1. the receiving process delays until a message with value MTEXT has been delivered such that $\beta^m_{\mathrm{MTEXT}} = TRUE$.

2. the program variable m is then assigned the value MTEXT.

This **receive** is *synchronous*; that is, the invoking process is delayed until an appropriate message is delivered. It should be noted that the implementation of such a **receive** is, roughly speaking, as complicated as β is allowed to be. Allowing unrestricted conditions β would be quite difficult, or impossible, to implement. Languages such as Ada, CSP, and SR [1] place restrictions on β to allow greater functionality in their respective **receive** commands. In our case, we assume that β=$TRUE$. This implies that no condition can be imposed on the message to be received for our particular **receive** statement.

4.2 Schlichting's Semantics

In this section, we summarize the semantics of **send** and **receive** as they appear in [24]. No attempt will be made to derive these rules, but some motivation will be given. The interested reader should refer to [24] for further details.

For any process D, the system state must include information about messages which have been sent to D by other processes, and about those messages which D has received. Associated with each process D are two *bags* or *multisets* which model this aspect of the system state. These bags are referred to as the *send* bag and *receive* bag of D, respectively. *Auxiliary* or *ghost* variables σ_D and ρ_D represent the send and receive bags, respectively, in the semantics. σ_D contains a copy of each message sent to D, while ρ_D contains a copy of each message received by D. Thus, $\rho_D \subseteq \sigma_D$. Additionally, we use bag *addition* $\oplus$ and *subtraction* $\ominus$ to express these semantics. For bag σ and object x, $\sigma \oplus x$ is the bag which contains one more copy of x and is otherwise identical to σ. For bag σ and object x, $\sigma \ominus x$ is the bag which contains one less copy of x, if $x \in \sigma$, and is otherwise identical to σ. $\sigma \ominus x = \sigma$ if $x \notin \sigma$.

When one process sends a message with value *expr* to process D, the result is an assignment to the send bag σ_D. Thus the axiom schema for **send** derives from the standard assignment axiom:

$$\{\, W^{\sigma_D}_{\sigma_D \oplus expr} \,\}\ \textbf{send}\ \ expr\ \textbf{to}\ D\ \ \{\, W \,\} \tag{4.1}$$

The axiom schema for the **receive** command in [24] is:

$$\{\, R \,\}\ \ r : \textbf{receive}\ \ m\ \textbf{when}\ \beta\ \ \{\, Q \,\} \tag{4.2}$$

provided that the following sentence is a theorem:

$$(\ R\ \wedge\ \beta^m_{MTEXT}\ \wedge\ MTEXT \in (\sigma_D\ \ominus\ \rho_D)) \implies Q^{m\ ,\ \rho_D}_{MTEXT, \rho_D \oplus MTEXT} \tag{4.3}$$

where D is the receiving process. It is important to note that in this assertion *MTEXT* is a *variable* and *not* a literal value (e.g., "16"). We will refer to this last sentence as $Sat(r, R, Q, m, \beta)$, or more simply, as $Sat(r)$. A proof of $Sat(r, R, Q, m, \beta)$ guarantees the *soundness* of an application of Schema (4.2).

Schema (4.2) would is clearly unsound by itself; assign the precondition $TRUE$ and the postcondition $FALSE$, for instance. However, if $Sat(r, R, Q, m)$ has been demonstrated, applying Schema (4.2) would be sound. Note that if the hypothesis of (4.3) is true, then there is a message $MTEXT$ available for process D to receive (since $MTEXT \in (\sigma_D \ominus \rho_D)$). When the receive statement r picks one such $MTEXT$, that $MTEXT$ is added to the receive bag ρ_D and assigned to program variable m. The other components of the state remain unchanged. Because $Sat(r, R, Q, m)$ has been demonstrated, $Q^{m\ ,\ \rho_D}_{MTEXT, \rho_D \oplus MTEXT}$ is true. Thus, postcondition Q is true in the state resulting from the execution of r.

In general, one would also require some means of demonstrating the *non-interference* of concurrent processes. That is, one needs some way of showing that assignment actions in one process do not interfere with the *critical assertions*[1] of other concurrently executing processes. However, for reasons that will be explained in the next chapter, non-interference need not be demonstrated for our system, so the details of showing non-interference for **send** and **receive** are not given here.

[1] For a rigorous definition of *critical assertion* and *process interference*, please refer to [1].

Chapter 5

The Target Language—Its Origins and Semantics

Currently in the UC Davis Laboratory for System Specification and Verification, we are designing, specifying, and verifying an operating system kernel which supports distributed computing through asynchronous message passing. We are extending the uniprocessor operating system kernel KIT, which was designed and verified at the University of Texas at Austin [3], to include kernel primitives for handling asynchronous **send** and **receive**. The kernel is to be written in the instruction set of the AVM-1 microprocessor, which was designed and verified by Phillip Windley at UC Davis [28]. The overall architecture of the distributed system is based upon the UNIX United system [23] developed by John Rushby and Brian Randell at the University of Newcastle upon Tyne.

The next three sections of this chapter focus on the properties of this distributed system which are relevant to our target language. A brief description of the components of UNIX United relevant to the target language is given — specifically, the assumptions about the network interface to the operating system. Our distributed system is based in part on UNIX United, and so a brief description is given as motivation. The KIT process model and its consequences for our language are discussed. In particular, demonstrating the *non-interference* of concurrent processes is shown to be unnecessary for our process model. The

AVM-1 microprocessor and the instructions which we wish to model are described. These instructions are, strictly speaking, *based* on AVM-1 instructions, and the assumptions made about these commands are explained. Only four commands will be mechanized in our logic, primarily because we wished to develop the *means* for reasoning about code in HOL, rather than the means for reasoning about programs in a particular programming language. Finally, we describe the target language, its axiomatic semantics, and the rules of inference for our formal system. We refer to our target language as L and its axiomatic semantics as the programming logic $\mathcal{PL}$.

5.1 The UNIX United Distributed System

Our distributed system DKIT is based on the UNIX United secure distributed system [23] designed by John Rushby and Brian Randell at the University of Manchester. We include a brief description of the UNIX United system because the "UCD Tower" of which DKIT is a part will be based in part on this system model.

A *multi-level security policy* (MLS) assigns each user and data object a security level from a partially ordered set of security classifications. A user should not gain access to data which has a security level "higher" than his own. Suppose, for example, that users are partitioned into two security levels: "unclassified" and "secret", then, an unclassified user should not gain access to a secret file, but a secret user may access either secret or unclassified data.

The UNIX United distributed system [23] attempts to enforce MLS by physically separating hosts of different security levels, and mediating communication across the local area network (LAN), as illustrated in Figure 5.1. A UNIX United distributed system (UUDS) consists of a collection of single-level hosts (i.e., a host where all users and data have the same security level) which are connected to a LAN. Distributed processes of the same security level may communicate through the LAN, and access to a host is controlled by a *trusted network interface unit* (TNIU). A TNIU resides between each host and the LAN. MLS is enforced by the TNIU, which only receives datagrams of security level identical to that of its host. The trusted network interface units are the only parts of this system which enforce security—the hosts themselves are untrusted.

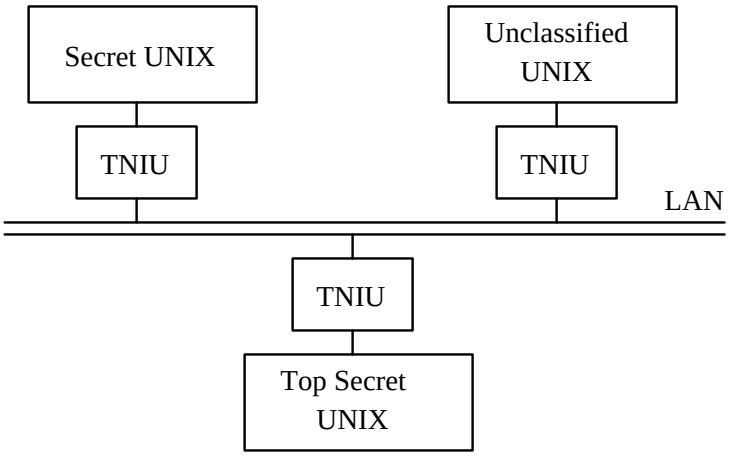

Figure 5.1: The UNIX United Distributed System

5.2 The Process Model

Our operating system kernel is based on the small uniprocessor kernel KIT (*Kernel for Isolated Tasks*). KIT provides multitasking and single-word message passing among processes. KIT also provides the following verified services [4]:

- "round-robin" process scheduling
- interrupt handling

However, KIT does not provide:

- dynamic process creation
- shared memory

KIT processes communicate *only* through message passing.

The main goal of the KIT project at UT Austin required the verification of *process isolation*, even in a setting involving asynchronous interrupts. Process isolation implies that the execution of one process can not change the state of another process except through

prescribed means (e.g., through message passing). So, for example, one process should not be able to overwrite another process's address space. It was demonstrated, among other things, that "the (KIT) context switch operation on an interrupt correctly maintains the state of all processes" [4]. We shall see later in this chapter that this is a particularly important property for avoiding process interference.

We assume that the kernel for our distributed system (for lack of a better term, it will be referred to from now on as DKIT) has the following KIT-like services and properties:

- multitasking
- no shared memory
- process communication through message passing only
- no dynamic creation of processes
- a context switch operation which correctly maintains the state of all processes
- static naming of processes

We assume further that message passing can take place between any two processes—whether or not they exist on the same network node. That is, process "1" in Figure 5.2 can send a message to process "2" or to process "a". Therefore, DKIT must extend the existing message passing service of KIT to allow datagrams to be exchanged between network nodes as well. The actual distribution of processes is completely transparent to processes, so a process in Figure 5.2 does not "know" on which network node it is executing.

We assume there are two DKIT kernel subroutines—**send** and **receive**—which implement asynchronous send and synchronous receive. The format of **send** and **receive** and their semantics will be provided in the last section of this chapter. We assume that **send** and **receive** execute *atomically*; that is, the subroutines execute without interruption until complete[1]. **send** and **receive** will be considered as commands in our target language L. Although KIT was written for the FM8501 microprocessor[18], we assume DKIT is written for the AVM-1 microprocessor.

[1] This is also important for avoiding process interference.

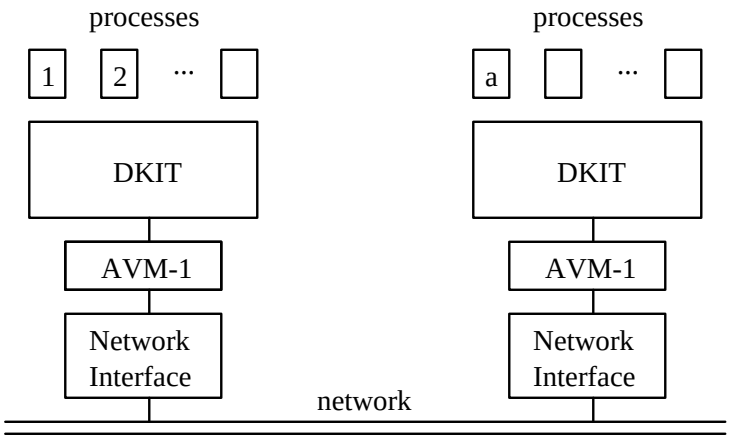

Figure 5.2: The Distributed System

5.3 The AVM-1 Microprocessor

As part of his doctoral dissertation [28] at UC Davis, Windley designed, specified, and verified the correctness of the AVM-1 (*A Verified Microprocessor*). AVM-1 has a load-store architecture and is, in most respects, a RISC machine. The instruction set contains thirty instructions, four of which will be mechanized in our logic.

L contains four AVM-1 instructions: `ADDI` (add immediate), `SUB` (subtract), `JMP_NZ` (jump if not zero), and `JMP_GE` (jump if greater than or equal to). We assume there are two flags — **NZ** and **GE** —which are set if the last arithmetic operation produced a non-zero result or a result greater than or equal to zero, respectively. Register transfer level (RTL) descriptions of `ADDI` and `SUB` can be found in Figures 5.3 and 5.4, respectively. In the RTL notation, we note that the assignments occurring in Figures 5.3 and 5.4 take place simultaneously. That is necessary in Figure 5.3 for example, when R_{dst}=R_{src}.

Strictly speaking, these instructions are AVM-1-*like*. We ignore *carry* and *overflow* arithmetic conditions, for example. Furthermore, it was convenient to model forward (backward) jumps with IF (WHILE) statements, because otherwise an axiomatic semantics for jumps would have to distinguish between the notions "forward" and "backward". It is difficult to express the axiomatic semantics of the **goto** statement as a *triple*—the existence of more

`ADDI`

$$
\begin{aligned}
R_{dst} &\Leftarrow R_{src} + imm \\
\mathbf{NZ} &\Leftarrow (R_{src} + imm) \neq 0 \\
\mathbf{GE} &\Leftarrow (R_{src} + imm) \geq 0 \\
PC &\Leftarrow PC + 1
\end{aligned}
$$

Figure 5.3: RTL for `ADDI` R_{src}, R_{dst}, imm

`SUB`

$$
\begin{aligned}
R_{dst} &\Leftarrow R_{src1} - R_{src2} \\
\mathbf{NZ} &\Leftarrow (R_{src1} - R_{src2}) \neq 0 \\
\mathbf{GE} &\Leftarrow (R_{src1} - R_{src2}) \geq 0 \\
PC &\Leftarrow PC + 1
\end{aligned}
$$

Figure 5.4: RTL for `SUB` R_{src1}, R_{src2}, R_{dst}

than one point of exit from a statement S is more easily expressed as a *Hoare n-tuple* with $n > 3$. [2] A general Hoare n-tuple semantics would be difficult to mechanize in HOL because HOL predicates have fixed arity. Because the IF statements and WHILE loops give the language sufficient expressive power for our purposes, and because no easily-mechanized, general rules for **goto** could be found, we settled for a more limited language.

Our language differs further from *real* AVM-1 in that instruction operands are assumed to have type *integer*. Consequently, the distinction between signed and unsigned integers, as well as the finite range of operands, are ignored. These simplifications were made because developing the *means* of reasoning in HOL about code was the primary goal of this project. Using the methods developed in this thesis, an axiomatic semantics for a language closer to AVM-1 can be mechanized.

5.4 The Target Language L and Its Semantics

Our target language L contains two arithmetic instructions— `ADDI` and `SUB` —which are based on AVM-1 instructions of the same name. L also contains the standard imperative programming language control structures IF and WHILE. Finally, L has **send** and **receive** commands, the semantics of which are derived from those described in Chapter 4.

The format of the IF statement in our language is:

$$\textbf{IF}\ \textit{boolean expression}\ \textbf{THEN}\ \textit{StatementList}\ \textbf{ENDIF} \tag{5.1}$$

Notice that there is no ELSE clause for our IF statement. The WHILE statement is standard. Its format is:

$$\textbf{WHILE}\ \textit{boolean expression}\ \textbf{DO}\ \textit{StatementList}\ \textbf{ENDWHILE} \tag{5.2}$$

The **send** command performs an asynchronous send. The format of the **send** command is:

$$\textbf{send}\ \mathtt{R}_{msg},\ dst \tag{5.3}$$

[2] For further discussion, see Arbib [2], Clint and Hoare [8], and Kowaltowski [19].

Executing the above statement sends a one word message containing the contents of register $\mathtt{R}_{msg}$ to the process named by the *dst* (recall that processes are statically named). If *dst* refers to an invalid name[3], **send** $\mathtt{R}_{msg}$, *dst* does nothing. **send** is *asynchronous*—the process executing **send** does not block after executing **send** .

The format of the **receive** command given in [24] is:

$$\textbf{receive } m \textbf{ when } \beta \tag{5.4}$$

To simplify our mechanization, we will assume for our **receive** command that β is always TRUE. The format for our **receive** command is then:

$$\textbf{receive } \mathtt{R}_{msg} \tag{5.5}$$

The process executing this command has the value of a (previously un-**receive**'d) message placed in $\mathtt{R}_{msg}$, if one such message is available. If such a message is not available, the receiving process blocks execution until a message arrives. **receive** implements *synchronous* receive. Because β is unspecified for general **receive** statements, a variety of communications primitives can be modeled—among them receive statements for named channels [24]. Messages could have a "sender" field which identifies the sending process, and if receiving condition β were "sender field is 5", that **receive** would only accept messages from process 5. But because β is assumed to be the constant TRUE, processes can not put any condition on which message to receive. For instance, process 1 can not distinguish as part of deciding whether to receive a message between messages sent to it by processes 2 and 3—even if the identity of the sending process is encoded into the message.

5.4.1 The Axiom Schemata

The axiomatic semantics for L can now be given. Axiom schemata for `ADDI` and `SUB` derive from the standard assignment axiom [1, 16, 21] in a straightforward manner. The axiom schemata for **send** and **receive** come directly from the semantics described in [24]

[3] *How* the processes are named in our abstract language is ignored. It suffices to know that each process has a unique name. We also leave unspecified what *dst* is. It could be a register name or an address which is ultimately interpreted as a process name.

which were outlined in chapter 4. The semantics for IF and WHILE are simplifications of those found in [1]. The semantics of IF and WHILE are inference rules, because their consequents depend upon the derivability of other triples in their antecedents, and so they will be described in the following section on inference rules.

The axiomatic semantics of an assignment statement "x := e" is usually given by the following schema:

$$\{\ \phi^{x}_{expr}\ \}\ \ x\ :=\ expr\ \{\ \phi\ \}.$$

Both ADDI and SUB are assignment-type statements; each assign a new value to $\mathtt{R}_{dst}$. ADDI $\mathtt{R}_{src}$, $\mathtt{R}_{dst}$, imm performs the assignment:

$$\mathtt{R}_{dst}\ \Leftarrow\ \mathtt{R}_{src}\ +\ imm.$$

Thus, one preliminary schema for ADDI is:

$$\{\ \phi^{\mathtt{R}_{dst}}_{\mathtt{R}_{src}\ +\ imm}\ \}\ \ \mathtt{ADDI}\ \ \mathtt{R}_{src},\ \mathtt{R}_{dst},\ imm\ \ \{\ \phi\ \}$$

However, ADDI also affects the condition flags **NZ** and **GE** . Specifically, **NZ** = $((\mathtt{R}_{src}\ +\ imm)\ \neq\ 0)$, and **GE** = $((\mathtt{R}_{src}\ +\ imm)\ \geq\ 0)$[4]. So the axiom schema for ADDI is:

$$\{\ \phi^{\mathtt{R}_{dst}}_{\mathtt{R}_{src}\ +\ imm}\ \}\ \ \mathtt{ADDI}\ \ \mathtt{R}_{src},\ \mathtt{R}_{dst},\ imm\ \ \{\ \phi\ \wedge\ CF_{\mathtt{ADDI}}\ \} \tag{5.6}$$

where $CF_{\mathtt{ADDI}}$ is

$$\mathbf{NZ}\ =\ (\mathtt{R}_{src}\ +\ imm\ \neq\ 0)\ \ \wedge\ \ \mathbf{GE} = (\mathtt{R}_{src}\ +\ imm\ \geq\ 0).$$

The axiom schema for SUB is derived from the standard assignment axiom similarly to the case for ADDI . It is:

$$\{\ \phi^{\mathtt{R}_{dst}}_{\mathtt{R}_{src1}\ -\ \mathtt{R}_{src2}}\ \}\ \ \mathtt{SUB}\ \ \mathtt{R}_{src1},\ \mathtt{R}_{src2},\ \mathtt{R}_{dst}\ \ \{\phi\ \wedge\ CF_{\mathtt{SUB}}\} \tag{5.7}$$

where $CF_{\mathtt{SUB}}$ is defined as

$$\mathbf{NZ}\ =\ (\mathtt{R}_{src1}\ -\ \mathtt{R}_{src2}\ \neq\ 0)\ \ \wedge\ \ \mathbf{GE} = (\mathtt{R}_{src1}\ -\ \mathtt{R}_{src2}\ \geq\ 0).$$

[4]This is technically an abuse of language— **NZ** and **GE** are *bits*, not booleans. The meaning should be clear, however.

Recall from schema (4.1) that the axiom schema for **send** in [24] is:

$$\{\ W^{\sigma_{dst}}_{\sigma_{dst} \oplus expr}\}\ \textbf{send}\ \ expr\ \textbf{to}\ dst\ \ \{\ W\ \}$$

where the destination process has name dst. Recall from section 4.2 that a **send** is an assignment to the send bag σ_{dst} of process dst. Since the expression $expr$ we wish to send to process dst is the contents of the *sender's* register $\mathtt{R}_{msg}$, the axiom schema for our **send** command is:

$$\{\ W^{\sigma_{dst}}_{\sigma_{dst} \oplus \mathtt{R}_{msg}}\}\ \textbf{send}\ \ \mathtt{R}_{msg},\ dst\ \ \{\ W\ \}. \tag{5.8}$$

Recall from schema (4.2) that the axiom schema for the **receive** command in [24] is:

$$\{\ R\ \}\ \ r : \textbf{receive}\ \ m\ \textbf{when}\ \beta\ \ \{\ Q\ \}$$

provided that the following sentence is a theorem:

$$(\ R\ \wedge\ \beta^{m}_{MTEXT}\ \wedge\ MTEXT \in (\sigma_D\ \ominus\ \rho_D))\ \Longrightarrow\ Q^{m\ ,\ \rho_D}_{MTEXT, \rho_D \oplus MTEXT}$$

where D is the receiving process. Recall that this sentence is referred to as $Sat(r, R, Q, m, \beta)$. In our case (since we assume β is $TRUE$), this rule simplifies to:

$$\{\ R\ \}\ \ r : \textbf{receive}\ \ m\ \ \{\ Q\ \}$$

whenever:

$$(\ R\ \wedge\ MTEXT \in (\sigma_D\ \ominus\ \rho_D))\ \Longrightarrow\ Q^{m\ ,\ \rho_D}_{MTEXT, \rho_D \oplus MTEXT}$$

So the axiom schema[5] for our **receive** command is:

$$\frac{\vdash\ \ Sat(\ r\ ,\ \phi\ ,\ \psi\ ,\ \mathtt{R}_{msg},\ TRUE\ )}{\{\ \phi\ \}\ \ r :\ \textbf{receive}\ \ \mathtt{R}_{msg}\ \ \{\ \psi\ \}}. \tag{5.9}$$

Axiom schemata (5.1)-(5.4) are summarized in Figure 5.5.

We have as yet no explicit representation of processes in our programming logic. Frequently, a *cobegin* statement represents the creation of some number of asynchronously executing processes as in [1, 21]. A process in our logic is identified with a code listing and a set of registers.

[5] The rule for **receive** is an axiom schema rather than a inference rule since it does not depend on any theorem in our *programming* logic (although it does depend on a theorem in our *assertion* logic).

Instruction	Semantics
ADDI $\mathtt{R}_{src}$, $\mathtt{R}_{dst}$, imm	$\{\ \phi^{\mathtt{R}_{dst}}_{\mathtt{R}_{src}+imm}\ \}$ ADDI $\mathtt{R}_{src}$, $\mathtt{R}_{dst}$, imm $\{\ \phi \wedge CF_{\mathtt{ADDI}}\ \}$
SUB $\mathtt{R}_{src1}$, $\mathtt{R}_{src2}$, $\mathtt{R}_{dst}$	$\{\ \phi^{\mathtt{R}_{dst}}_{\mathtt{R}_{src1}-\mathtt{R}_{src2}}\ \}$ SUB $\mathtt{R}_{src1}$, $\mathtt{R}_{src2}$, $\mathtt{R}_{dst}$ $\{\ \phi \wedge CF_{\mathtt{SUB}}\ \}$
send $\mathtt{R}_{src}$, dst	$\{\ W^{\sigma_{dst}}_{\sigma_{dst}\oplus\mathtt{R}_{msg}}\}$ **send** $\mathtt{R}_{msg}$, dst $\{\ W\ \}$
receive $\mathtt{R}_{msg}$	$\frac{\vdash\ Sat(\ r\ ,\ \phi\ ,\ \psi,\ \mathtt{R}_{msg},\ TRUE\ )}{\{\ \phi\ \}\quad r\text{: }\mathbf{receive}\ \mathtt{R}_{msg}\quad \{\ \psi\ \}}$

Figure 5.5: The Instruction Mnemonics and Their Semantics

5.4.2 Non-interference

When two or more processes write to the same program variable, the results of the program can be *non-deterministic* if more than one outcome may result from the execution. Consider two processes each executing "**x** := **x** + 1" where **x**=0 initially. Here, the two processes perform the following steps:

(1.) read the value of **x**,
(2.) add 1 to that value, and then
(3.) write the incremented value back to the address of **x**.

The time at which each process executes (1.) through (3.) affects the final value of **x**. At the end of execution, the value of **x** may be either 1 or 2. The two processes above may then *interfere* with one another. It is necessary to demonstrate the *non-interference* of processes for our concurrent language.

For our language, non-interference of processes follows from the assumed properties of the underlying distributed system in which our code executes. In Section 5.1, we assumed that in the distributed system:

- processes do not share memory,
- the context switch operation correctly maintains the state of all processes—that is, the state of a process—registers, flags, and memory—remains unaffected by context switching,
- the DKIT kernel calls **send** and **receive** execute without interference.

The instructions `ADDI` and `SUB` are machine instructions, which execute atomically. Since processes do not share memory and context switches are process-transparent, the execution of these instructions can not affect or be affected by another process. Our semantics for **send** and **receive** include *non-interference* assumptions, stating that the execution of these statements are atomic with respect to the objects they manipulate. The objects in question are machine registers and the send and receive bags, which record the messages sent to and received by a process, respectively. These assumptions would incur proof obligations on a verified implementation of **send** and **receive**.

5.4.3 The Inference Rules

The rules of inference for the programming logic $\mathcal{PL}$ are standard for any programming language with IF and WHILE. The rules for IF and WHILE derive from the more complex rules for concurrent (multi-arm) IF and WHILE found in [1]. Because these rules are so standard, explanation of them will not be given here. However, those interested may find expositions in [1, 16, 21]. The rules for $\mathcal{PL}$ are summarized in Figure 5.6.

Axiom Introduction:

$$\frac{\{\,\phi\,\}\ SL\ \{\,\psi\,\}\ \textit{is an axiom}}{\vdash_{\mathcal{PL}}\ \{\,\phi\,\}\ SL\ \{\,\psi\,\}}$$

Composition Rule:

$$\frac{\vdash_{\mathcal{PL}}\ \{\,\phi\,\}\ SL_1\ \{\,\psi\,\},\ \vdash_{\mathcal{PL}}\ \{\,\gamma\,\}\ SL_2\ \{\,\theta\,\},\ \vdash\ \psi \Longrightarrow \gamma}{\vdash_{\mathcal{PL}}\ \{\,\phi\,\}\ SL_1; SL_2\ \{\,\theta\,\}}$$

Left Consequence:

$$\frac{\vdash_{\mathcal{PL}}\ \{\,\phi\,\}\ SL\ \{\,\psi\,\},\ \vdash\ \gamma \Longrightarrow \phi}{\vdash_{\mathcal{PL}}\ \{\,\gamma\,\}\ SL\ \{\,\psi\,\}}$$

Right Consequence:

$$\frac{\vdash_{\mathcal{PL}}\ \{\,\phi\,\}\ SL\ \{\,\psi\,\},\ \vdash\ \psi \Longrightarrow \gamma}{\vdash_{\mathcal{PL}}\ \{\,\phi\,\}\ SL\ \{\,\gamma\,\}}$$

IF rule:

$$\frac{\vdash_{\mathcal{PL}}\ \{\,\phi \wedge \mathrm{B}\,\}\ SL\ \{\,\psi\,\},\ \vdash\ (\phi \wedge \neg \mathrm{B}) \Longrightarrow \psi}{\vdash_{\mathcal{PL}}\ \{\,\phi\,\}\ \textbf{if}\ \mathrm{B}\ \textbf{then}\ SL\ \textbf{endif}\ \{\,\psi\,\}}$$

WHILE rule:

$$\frac{\vdash_{\mathcal{PL}}\ \{\,\mathrm{Inv} \wedge \mathrm{B}\,\}\ SL\ \{\,\mathrm{Inv}\,\}}{\vdash_{\mathcal{PL}}\ \{\,Inv\,\}\ \textbf{while}\ \mathrm{B}\ \textbf{do}\ SL\ \textbf{endwhile}\ \{\,\mathrm{Inv} \wedge \neg \mathrm{B}\,\}}$$

Figure 5.6: Inference Rules for the Programming Logic $\mathcal{PL}$

Chapter 6

The HOL Mechanization of the Semantics

This chapter describes in detail how the HOL integration of the axiom schemata and inference rules of the language took place. A predicate `Prov` is defined inductively so that

$$\{\phi\}\ \ StatementList\ \ \{\psi\}$$

is a theorem in the logic if and only if

$$\texttt{Prov}\ \ \phi\ \ StatementList\ \ \psi$$

is a theorem of HOL. `Prov` is an abbreviation of "is provable". The pre- and postconditions of a Hoare triple are represented in HOL as terms of the type "bool", and *StatementList* has HOL type "(string)list" (i.e., a list of strings).

6.1 The Rules of Inference

We mechanize the inference rules of our system $\mathcal{PL}$ using the inductive relation definitions package of HOL[20] described in Section 3.3. The relation `Prov` is defined in Figure 6.1. The relation `Prov` has type "`:bool->(string)list->bool->bool`" and is defined inductively.

Each inductive step in Figure 6.1 specifies an inference rule. Consider the step labeled "Axiom introduction" in Figure 6.1:

```
[ [    "IS_AXIOM (t1:bool) (st:(string)list) (t2:bool):bool"] ,
      %-----------------------------------%
       "^Prov t1 st t2" ;
```

That is, for each triple satisfying `IS_AXIOM` ϕ *StatementList* ψ[1], we can conclude:

`Prov` ϕ *StatementList* ψ.

Thus, any axiom is a theorem of our system.

Another example of how the inference rules of the system work is the "Composition Rule":

```
[     "^Prov (t1:bool) (st1:(string)list) (t2:bool)";
      "^Prov (t3:bool) (st2:(string)list) (t4:bool)";
      "t2⟹ t3" ],
    %-------------------------------------%
      "^Prov t1 (APPEND st1 st2) t4" ;
```

If $\vdash$ `Prov t1 st1 t2`, $\vdash$ `Prov t3 st2 t4`, and $\vdash$ `t2` $\Longrightarrow$ `t3`, then:

$\vdash$ `Prov t1 st1^st2 t4`.

Here, the caret ^ stands for "list appending" and has nothing whatsoever to do with HOL antiquotation. The rules for left consequence ("precondition strengthening"), right consequence ("postcondition weakening"), and the IF and WHILE statements are defined similarly. The inductive definitions package provides a simple and elegant means of mechanizing the inference rules of our system.

The inductive definitions package returns:

[1] `IS_AXIOM` will be discussed later in this chapter. It means roughly "constructed by an axiom schema".

```
let (Prov_rules,Prov_ind) =
   let Prov = "Prov:bool->(string)list->bool->bool"
in
   new_inductive_definition false `Prov`
   ("^Prov t1 st t2", [])
%Axiom Introduction%
   [ [    "IS_AXIOM (t1:bool) (st:(string)list) (t2:bool):bool"] ,
          %-----------------------------------%
          "^Prov t1 st t2" ;
%Composition Rule%
     [     "^Prov (t1:bool) (st1:(string)list) (t2:bool)";
           "^Prov (t3:bool) (st2:(string)list) (t4:bool)";
           "t2 ⟹  t3" ],
         %-----------------------------------%
           "^Prov t1 (APPEND st1 st2) t4" ;
%Left Consequence%
     [     "^Prov (pre:bool) (st:(string)list) (post:bool)";
           "phi ⟹  pre"],
         %-----------------------------------%
           "^Prov phi st post";
%Right Consequence%
     [     "^Prov (pre:bool) (st:(string)list) (post:bool)";
           "post ⟹  phi"],
         %-----------------------------------%
           "^Prov pre st phi";
%IF rule for forward jumps%
     [     "^Prov (P ∧ B) (st:(string)list) (Q:bool)";
           "P ∧ ¬B ⟹ Q" ],
         %-----------------------------------%
           "^Prov P (APPEND [` IF `] (APPEND st [` ENDIF `] )) Q" ;
%WHILE rule for backward jumps%
     [     "^Prov (Inv ∧ B) (st:(string)list) (Inv:bool)" ],
         %-----------------------------------%
           "^Prov (Inv)
            (APPEND [` WHILE `] (APPEND st [` ENDWHILE `]))
                                              (Inv ∧ ¬B)"  ];;
```

Figure 6.1: Code for `Prov`

```
⊢ ∀t1 st1 st2 t4.
    (∃t2 t3. Prov t1 st1 t2 ∧ Prov t3 st2 t4 ∧ (t2 ⟹  t3)) ⟹
    Prov t1(APPEND st1 st2)t4;
```

for the composition rule. One could use the rule in this form every time one wishes to compose two triples; however, this is inconvenient and awkward, especially to those who are accustomed to constructing proofs on paper. To avoid burdening the user with unnecessary and confusing implementation details, two ML functions—`compose` and `help_compose`—have been written to provide a more natural proof environment. For each of the inference rules, there are analogous ML functions which allow proof construction in this HOL theory to appear very similar to a "hand" proof.

`compose` takes as arguments the three theorems necessary to apply the composition rule, namely:

- $\vdash$ `Prov` ϕ sl_1 ψ
- $\vdash$ `Prov` γ sl_2 θ
- $\vdash \psi \Longrightarrow \gamma$.

Given these three theorems, `compose` returns:

$$\vdash_{HOL} \quad \texttt{Prov}\ \phi\ \ sl_1\hat{}sl_2\ \ \theta$$

An example application of `compose` can be found in Figure 6.2. In that figure, the HOL and ML commands typed by the user are preceded by a "`#`", and the response of the HOL interpreter appears directly below each command. In Figure 6.2, the user first creates two axioms[2] (i.e., `th1` and `th2`), proves "`R2 = 2` $\Longrightarrow$ `R2 = 2`", and then applies `compose`, from which the desired theorem results. `compose` ensures that `th3` *really* is the necessary theorem by checking that the postcondition of `th1` is the antecedent of `th3`, and that the precondition of `th2` is the consequent of `th3`.

[2]Details of what is meant by this will be provided later in this chapter.

```
#new_constant(`R1`,":num");;
   () : void
#new_constant(`R2`,":num");;
   () : void
#let th1 = mk_ADDI "[`ADDI R1,R2,1`]" "R2 = 2" "R2" "R1" "1";;
   th1 = ⊢ Prov(R1 + 1 = 2)[`ADDI R1,R2,1`](R2 = 2)
#let th2 = mk_ADDI "[`ADDI R1,R1,1`]" "R2 = 2" "R1" "R1" "1";;
   th2 = ⊢ Prov(R2 = 2)[`ADDI R1,R1,1`](R2 = 2)
#let th3 = DISCH "R2 = 2" (ASSUME "R2 = 2");;
   th3 = ⊢ (R2 = 2) ⟹  (R2 = 2)
#compose th1 th2 th3;;
   ⊢ Prov(R1 + 1 = 2)[`ADDI R1,R2,1`;`ADDI R1,R1,1`](R2 = 2)
```

Figure 6.2: `compose` Example

The ML function `help_compose` takes the two theorems "⊢ `Prov` ϕ sl_1 ψ" and "⊢ `Prov` γ sl_2 θ", constructs the sentence "$\psi \Longrightarrow \gamma$", and sets it as an HOL proof *goal*. The tactics package can then be used to demonstrate " ⊢ $\psi \Longrightarrow \gamma$" (assuming, of course, that it really *is* a theorem). A sample application of `help_compose` can be found in Figure 6.3. Theorems `th1` and `th2` are identical to those in Figure 6.2. Instead of constructing the statement of `th3` by hand as in Figure 6.2, the user could use `help_compose` as in Figure 6.3. `help_compose` constructs the necessary sentence (in this case "`R2 = 2` $\Longrightarrow$ `R2 = 2`"), and sets it as a proof goal. The user then proves the goal using the HOL tactics package. Although this is a trivial example, `help_compose` becomes very useful when pre- and postconditions become complicated. We created functions similar to `compose` and `help_compose` for the rules of left and right consequence as well. These functions are named `l_conseq`, `help_l_conseq`, `r_conseq`, `help_r_conseq`, and are applied in a manner completely analogous to the examples found in Figures 6.2 and 6.3.

```
#th1;;
   th1 = ⊢ Prov(R1 + 1 = 2)[‘ADDI R1,R2,1‘](R2 = 2)
#th2;;
   th2 = ⊢ Prov(R2 = 2)[‘ADDI R1,R1,1‘](R2 = 2)
#help_compose th1 th2;;
   "(R2 = 2) ⟹  (R2 = 2)"
#e(DISCH_TAC THEN ASM_REWRITE_TAC []);;
   OK..
   goal proved
   ⊢ (R2 = 2) ⟹  (R2 = 2)
#let th3 = top_thm();;
   th3 = ⊢ (R2 = 2) ⟹  (R2 = 2)
```

Figure 6.3: `help_compose` Example

```
#new_constant(`R1`,":num");;
   () : void
#new_constant(`R2`,":num");;
   () : void
#let th = mk_ADDI "[`ADDI R1,R2,1`]" "R2 = 2" "R2" "R1" "1";;
   th = ⊢ Prov(R1 + 1 = 2)[`ADDI R1,R2,1`](R2 = 2)
```

Figure 6.4: `mk_ADDI` Example

6.2 The Axiom Schemata

For each axiom schema, we provide an ML function has been provided which will create an instance of that axiom. The simplest axiom schemata are for the instructions `ADDI`, `SUB`, and `send` because the semantics of these instructions derive from the standard assignment axiom of Hoare semantics. The ML functions associated with these schemata—`mk_ADDI`, `mk_SUB`, and `mk_send` — are therefore quite similar and simple to apply. The necessity of demonstrating soundness complicates the schema for `receive`, and the associated ML function—`mk_receive` — is somewhat more difficult to use than the others. However, a function `mk_Sat_rec` can assist in the demonstration of soundness.

`mk_ADDI` has type "`:(string)list -> bool -> num -> num -> num -> thm`", where the first, second, and third "`num`" components are for the destination, source, and immediate value of the `ADDI` instruction. The sample application of `mk_ADDI` found in Figure 6.4 will illuminate this point. Here, the user specifies the statement form "`[`ADDI R1,R2,1`]`", the postcondition "`R2 = 2`", the destination "`R2`", the source "`R1`", and `mk_ADDI` constructs the correct precondition "`R1 + 1 = 2`" from the postcondition, and returns a theorem stating that the corresponding triple satisfies `Prov`. The construction of these preconditions uses ML functions for simultaneous substitution which we wrote ourselves.

The ML function `mk_SUB` performs similarly to `mk_ADDI`. `mk_SUB` has type "`:(string)list`

```
#new_constant(‘R3‘,":num");;
() : void
#let th = mk_SUB "[‘SUB R1,R2,R3‘]" "R3 = 2" "R2" "R1" "R3";;
th =
⊢ Prov
   (R1 − R2 = 2)
   [‘SUB R1,R2,R3‘]
   ((R3 = 2) ∧ (NZ = ¬(R1 = R2)) ∧ (GE = R1 ≥ R2))
```

Figure 6.5: `mk_SUB` Example

`-> bool -> num -> num -> num -> thm`". A sample application of `mk_SUB` can be found in Figure 6.5. Here, the user specifies the instruction mnemonic "`[‘SUB R1,R2,R3‘]`", the postcondition "`R3 = 2`", the subtrahend "`R2`", the minuend "`R1`", and the destination "`R3`", and then `mk_SUB` constructs the correct precondition "`R1 - R2 = 2`". `mk_SUB` then proves that the corresponding triple satisfies `Prov`, and returns the resulting theorem. Note that the flags `NZ` and `GE` are set by this operation.

`mk_send` has type "`:bool -> num -> (num)bag -> thm`". The user provides the postcondition, the message value, and the *send bag* of the destination process. `mk_send` constructs the correct precondition and proves that the corresponding triple satisfies `Prov`. A sample application of `mk_send` can be found in Figure 6.6.

The last of the axiom schemata for the target language is for the `receive` command. The necessity of demonstrating *Sat(r)* for every `receive` statement r complicates the use of `mk_receive`. The user must prove $Sat(r)$ *before* applying `mk_receive`. The ML function `mk_receive` takes as arguments a theorem t which proves $Sat(r)$, the destination variable m, and the desired postcondition Q. In order to demonstrate the soundness of:

$$r \ : \ \{\, R \,\} \ \texttt{receive} \ m \ \ \{\, Q \,\}\,,$$

```
#let sigD_DEF = new_definition(‘sigD_DEF‘,
                "sigD = (ABS_bag λx:num. 0)");;

   sigD_DEF = ⊢ sigD = ABS_bag(λx. 0)

#let th = mk_send ‘‘[‘ send R1 to D ‘]"
"(16 IN_B sigD) ∧
 (sigD = (16 INSERT_B EMPTY_BAG)) ∧ (∀M:num. (M IN_B sigD) ⟹
(M = 16)) ∧
 (R1 = 16)"  "R1" "sigD";;

   th =
   ⊢ Prov
      (16 IN_B (R1 INSERT_B sigD) ∧
       (R1 INSERT_B sigD = 16 INSERT_B EMPTY_BAG) ∧
       (∀M. M IN_B (R1 INSERT_B sigD) ⟹  (M = 16)) ∧
       (R1 = 16))
       [‘ send R1 to D ‘]
      (16 IN_B sigD ∧
      (sigD = 16 INSERT_B EMPTY_BAG) ∧
      (∀M. M IN_B sigD ⟹  (M = 16)) ∧
      (R1 = 16))
```

Figure 6.6: `mk_send` Example

the following sentence

$$Sat(r) \; : \; R \;\; \wedge \;\; MTEXT \, \epsilon \, (\, \sigma_D \; \ominus \; \rho_D \,) \;\; \Longrightarrow \;\; Q^{m \, , \, \rho_D}_{MTEXT, \rho_D \oplus MTEXT}$$

must be shown to be a theorem. Given theorem t demonstrating $Sat(r)$, m, and Q, `mk_receive` verifies that t is in the correct form, extracts the precondition R from t, and then proves and returns the theorem:

$$\vdash \texttt{Prov} \;\; (R) \, [\text{`}\texttt{receive}\text{`}] \, (Q) \, .$$

An example application of `mk_receive` can be found in Figure 6.7. In this example, four new constants `x`, `PREC` ("*precondition*"), `POST` ("*postcondition*"), and `mess` ("*message*") are added to the language, and the ML function `mk_thm` is applied to yield a theorem for $Sat(r)$ where r is "`receive x`". `mk_thm` was used here *purely for didactic purposes.* `mk_thm` should **not** be used to prove $Sat(r)$. `mk_receive` checks that `test` is in the correct form for the above triple, then proves and returns the theorem stating that the above triple satisfies `Prov`.

Constructing the $Sat(r)$ sentence may lead to frustration and despair on the part of the user because one human error can result in considerable wasted time and effort. It is quite easy, for example, for a human to err while performing simultaneous substitution. The author experienced this frustration when first using `mk_receive`. To avoid unnecessary *Sturm und Drang*, an ML function called `mk_Sat_rec` was created. `mk_Sat_rec` takes the precondition R, the postcondition Q, the destination variable m, the message variable[3] $MTEXT$, the send and receive bags σ and ρ, and then constructs the correct satisfaction sentence, and sets the result as an HOL proof *goal*. The user may then attempt to prove the sentence with the HOL tactics package. Applying `mk_Sat_rec` is similar to using `help_compose`.

6.3 Discussion

Hoare triples can be viewed as *predicate transformers* [1]. A programming language command transforms its precondition into its postcondition in a manner which can be described

[3] Recall from Section 4.2 that it is important that $MTEXT$ be a *variable* and not a literal value.

```
#new_constant(‘x‘,":num");;
   () : void
#new_constant(‘PREC‘,":bool");;
   () : void
#new_constant(‘POST‘,":bool");;
   () : void
#new_constant(‘mess‘,":num");;
   () : void
#let test = mk_thm([],
          "PREC ∧ ((mess:num) IN_B (sigD DIFF_B rhoD)) ⟹  POST");;
   test = ⊢ PREC ∧ mess IN_B (sigD DIFF_B rhoD) ⟹  POST
#mk_receive "[‘ receive x ‘]" test "x" "POST";;
   ⊢ Prov PREC[‘ receive x ‘]POST
```

Figure 6.7: `mk_receive` Example

```
#mk_Sat_rec
   "(rhoD = EMPTY_BAG) ∧ (∀M:num. (M IN_B sigD) ⟹  (M = 16))"
   "(R_D=16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))"
   "R_D" "MTEXT" "sigD:(num)bag" "rhoD:(num)bag";;


   "((rhoD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16))) ∧
    MTEXT IN_B (sigD DIFF_B rhoD) ⟹
    (MTEXT = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))"
```

Figure 6.8: `mk_Sat_rec` Example

syntactically. In this mechanization, ML functions were used to *directly* implement the Hoare rules (read "transformations") by transforming HOL boolean terms with (for the most part) pre-existing ML functions (e.g., `mk_conj`). It was discovered that the only ML function needed which was missing from ML was one to perform simultaneous substitution on terms. All the functions associated with axiom schemata—`mk_ADDI`, `mk_SUB`, `mk_send`, and `mk_receive`—were implemented this way.

It is impossible (or at least, beyond the author's capability) to reason about the syntactic structure of a term in HOL. For instance, one can not define an HOL predicate `IS_SIM_SUB` such that:

$$\vdash_{HOL} \texttt{IS\_SIM\_SUB}\ (\phi : term)\ (\psi : term)\ (x : num)\ (expr : num)$$

if and only if $\phi = \psi^{x}_{expr}$. One suggested alternative to the mechanization approach in this thesis is to define a "metalogic" of HOL which mirrors the HOL logic. One could define the expressions of this "metaHOL" logic in terms of trees, for example. Reasoning about the syntax of metaHOL would then be reasoning about parse trees. Clearly, `IS_SIM_SUB` could be defined for *metaHOL* terms. The preconditions and postconditions of a triple would then be represented as boolean metaHOL terms. This approach is perfectly valid, and one could certainly implement the semantics of the target language this way.

This approach has severe pitfalls, however. For example, suppose one needs to use the fact that $x + y = y + x$ for $x, y \in \mathbf{Z}$. This is a pre-proved theorem in HOL. It is *not* a theorem in metaHOL—or at least not until *you* prove it. Every theorem of HOL desired by the metaHOL user would have to be re-proved. This, in itself, is a considerable amount of work — work which, incidentally, has little or nothing to do with code verification. The (herculean) task of re-proving such theorems would greatly hinder the speed of the verification effort.

All of this begs the question of what a "proof" is in metaHOL. The primitive rules of inference of HOL would have to be defined for metaHOL to give meaning to the notion of "proof". Tactics would also have to be redesigned as well. Satisfactorily defining "proof" in metaHOL would involve a huge, tedious repetition of previously performed work.

The bags library in HOL was readily incorporated into the logic described in this thesis

simply by performing "`load_library 'bags';;`". So, Hoare rules involving bags (e.g., for **send**) can contain statements about bags and operations on bags, and pre-proven[4] results can be used. With our method, libraries defining new types and operations can, so to speak, be used "off the shelf". This would not be the case for metaHOL. As with the standard HOL libraries, any new library would have to be re-compiled in metaHOL. Again, the user would find himself engulfed by HOL system details which have little to do with verification.

The contrast between this approach and ours highlights the strengths of our method. They are:

- ease of modification—new rules can be readily added to the system.
- ease of use—help functions eliminate much user frustration and error, and the HOL tactics package can be applied directly to pre- and postconditions as needed. Also, large libraries of pre-proven theorems are available to the user in HOL.
- familiarity—someone completely innocent of HOL can look at an HOL session transcript of a derivation and understand what has been demonstrated.

The target language will certainly be expanded as necessary to complete the secure, distributed systems project, and the tools to do so have been developed and tested in this thesis. The help functions (e.g., `mk_Sat_rec`) free the user from performing tedious and complicated syntactic operations like simultaneous substitution, thereby eliminating much human error and effort. Proofs in this system are created and are represented in a manner similar to hand proofs, which renders them more accessible to the HOL novice —or to those lucky enough not to know HOL at all.

Our approach has two principal weaknesses—reliance on user-designed ML functions and `IS_AXIOM`. New axiom schemata may be implemented by the user, and may, therefore, introduce error. The user who introduces new axiom schemata into the system must take great care that his or her new ML code is correct. Furthermore, the ML functions for simultaneous substitution— `sim_sub` and `once_ssub`—could probably be written in a more

[4] That is, theorems already proven by someone else.

direct manner. The following is the ML code for `sim_sub`, which performs simultaneous substitution on HOL *terms* (rather than on theorems):

```
let sim_sub =
(λ wff:term. λ target:term. λ expr:term.
          snd( dest_thm (INST [(expr,target)] (mk_thm([],wff)))));;
```

The definition of `sim_sub` exploits the fact that simultaneous substitution for theorems can be performed with `INST`. The above definition converts the formula "`wff`" into a theorem using `mk_thm`, performs the desired substitution yielding another theorem, and then destroys the result to obtain a plain HOL term. The above definition is somewhat indirect, but it does work well, and more importantly, it **does not** add any additional strength to the theory.

The other weakness of the system is the use of `IS_AXIOM`. Consider the rule labeled "Axiom introduction" in Figure 6.1:

```
   [ [    "IS_AXIOM (t1:bool) (st:(string)list) (t2:bool):bool"] ,
        %-----------------------------------%
         "^Prov t1 st t2" ;
```

It was remarked earlier in this section that one can not reason in HOL about the *syntactic* structure of HOL terms. For this reason, `IS_AXIOM` can not enforce any requirements on the pre- and postconditions of a triple, and must, therefore, be defined as:

```
let IS_AXIOM = new_definition( `IS_AXIOM`,
        "∀(phi1:bool) (sl:(string)list) (phi2:bool).
                (IS_AXIOM phi1 sl phi2) = T");;
```

Thus, `IS_AXIOM` ϕ *StatementList* ψ is true for all ϕ, ψ, and *StatementList*. So literally for all triples { ϕ } *StatementList* { ψ },

$$\vdash_{HOL} \quad \texttt{Prov}\ \phi\ \mathit{StatementList}\ \psi$$

However, we restrict the use of the axiom introduction rule to those ML functions which implement the axiom schemata. If a user introduces axioms with axiom schema functions *exclusively*, the only triples derived will be those which follow from the axiomatic semantics of the target language. Thus, the user must be cautious when introducing new axiom schema functions. It is the opinion of the author that the caution which must be employed on the introduction of new axiom schema functions is a very small price to pay— particularly when considering the alternative.

The relative safety of a "pure" HOL semantics specification—like Gordon's — is greater than that of this system. We rely on new ML functions to perform syntactic operations like simultaneous substitution. These are potential sources of error. Of course, proofs in "pure" HOL rely on the correctness of certain ML functions, the ML interpreter, and the underlying architecture (among other things). We also rely on the user to apply the functions correctly. If for example one generates a *Sat(r)* sentence with `mk_Sat_rec` using a literal value (e.g., "16") for *MTEXT*, one can end up proving things which one should not prove. We have attempted to make assumptions of "good faith" on the part of the user only where an automated check could be made. It would be fairly easy to write a "good faith" checker which analysed the HOL code looking for literal values in the applications of `mk_Sat_rec`. Finally, some people are suspicious of axiomatic semantics which are not derived from a denotational model—and rightly so. Axiomatic semantics without a denotational underpinning *are* suspicious as one can not be truly sure that a language exists which satisfies those axioms without exhibiting a model. However, denotational models for asynchronous constructs are hard to define [25], so an axiomatic semantics may be the only one available.

Chapter 7

Example Proofs in $\mathcal{PL}$

In this chapter, two sample derivations in the system are presented—both taken from actual HOL sessions. In both examples, the HOL code typed by the user is preceded by a pound sign "#", and is followed by the responses of the HOL interpreter. The first example is presented with complete detail. Everything typed by the user and all responses of the HOL system are given, including the proofs of theorems necessary to apply rules of inference (e.g., composition, etc.). This was done to give the reader an appreciation for the "feel" of the system. However for the second, more difficult example, many of these details were suppressed for the sake of readability. The proofs of many of the theorems necessary for the application of inference rules in this example are long, and their presentation would not help new users of this system. The full HOL code listings can be found in Appendices B and C. Notationally, we represent a process in our logic as a code listing and a set of registers disjoint from the register set of any other process. We require that these register sets be disjoint to distinguish between the process or machine *contexts* of a register. So, if R_1 is used by two processes on one machine, *two* HOL constants are required to represent each context of R_1.

A ::

$\{\sigma_D = \Phi \ \wedge \ Inv \ \wedge \ R\_A\_16 = 16\}$

ADDI R_A_16, R_A, 0;

send R_A **to** D

$\{16 \in \sigma_D \ \wedge$

$\sigma_D = (\Phi \oplus 16) \ \wedge \ Inv \ \wedge$

$R\_A = 16\}$

D ::

$\{\rho_D = \Phi \ \wedge \ Inv\}$

receive R_D

$\{R\_D = 16 \ \wedge \ Inv\}$

where Inv is $(\forall M.\ M \in \sigma_D \implies (M = 16))$ and Φ is the empty bag.

Figure 7.1: A Simple Example

7.1 A Simple Example

Figure 7.1 contains a proof outline of the first example. There are two processes—"A" and "D". A sends a message containing the number 16 to process D. It is further assumed that only messages containing 16 are sent to D, and the invariant reflects this. It is demonstrated that, given these assumptions, the value of R_D after D receives is exactly 16.

In session box 1, the necessary definitions file "`hoare_defs.ml`" is loaded, the auxiliary variables for the send and receive bags ("`sigD'`" and "`rhoD`", respectively) and the registers `R_A`, `R_A_16`, and `R_D` are defined. Program and auxiliary variables (i.e., `R_A` and `sigD`) in HOL as constants in this mechanization. Usually program variables are referred to as "free" in Hoare logics, since they are never quantified over; however, in order to define simultaneous substitution in terms of pre-existing HOL and ML functions, it was convenient to represent

variables as HOL constants. These constants are really used as typed place-holders and the fact that they are constants in HOL is relevant only to the HOL realization of the axioms and inference rules. `sigD` and `rhoD` are defined as the empty bag (although this will not be clear to those not familiar with the HOL bag library). If a letter such as `A` or `D` occurs in the name of a register, then the name refers to the register in the context of process `A` or `D`, respectively.

1

```
#loadf `/usr/home/harrison/hol/hoare_defs.ml`;;

#let sigD_DEF = new_definition(`sigD_DEF`,
                "sigD = (ABS_bag λx:num. 0)");;
   sigD_DEF = ⊢ sigD = ABS_bag(λx. 0)
   Run time: 0.4s
   Intermediate theorems generated: 2

#let rhoD_DEF = new_definition(`rhoD_DEF`,
                "rhoD = (ABS_bag λx:num. 0)");;
   rhoD_DEF = ⊢ rhoD = ABS_bag(λx. 0)
   Run time: 0.3s
   Intermediate theorems generated: 2

#new_constant(`R_A`,":num");;
   () : void
   Run time: 0.0s

#new_constant(`R_A_16`,":num");;
   () : void
   Run time: 0.0s

#new_constant(`R_D`,":num");;
   () : void
   Run time: 0.0s

#new_constant(`MTEXT`,":num");;
   () : void
   Run time: 0.0s
```

In session box 2, the axiom schema for `mk_ADDI` is applied, returning the theorem called `th0` pictured in Figure 7.2. The `ADDI` command assigns the value of `R_A_16+0` to `R_A` and sets the condition flags *NZ* and *GE* appropriately. This command is, essentially, an

$$\{\sigma_D = \Phi \ \wedge \ Inv \ \wedge \ R\_A\_16 + 0 = 16 \ \wedge \ R\_A\_16 = 16\}$$
$$\texttt{ADDI}\ \ R\_A\_16,\ R\_A,\ 0;$$
$$\{\sigma_D = \Phi \ \wedge \ Inv \ \wedge \ R\_A = 16 \ \wedge \ R\_A\_16 = 16 \ \wedge$$
$$NZ = R\_A\_16 + 0 \neq 0 \ \wedge \ GE = (R\_A\_16 + 0) \geq 0\}$$

Figure 7.2: Proof Outline for HOL Session Box 2

assignment of `R_A_16+0` to `R_A`.

```
#let th0 =                                                              2
  let th = mk_ADDI "[‘ADDI R_A_16,R_A,0‘]"
   "(sigD = EMPTY_BAG) ∧ (∀M:num. (M IN_B sigD) ⟹
    (M = 16)) ∧ ((R_A=16) ∧ (R_A_16=16))"
         "R_A" "R_A_16" "0" in
                          REWRITE_RULE [] th;;

   th0 =
   ⊢ Prov
   ((sigD = EMPTY_BAG) ∧
    (∀M. M IN_B sigD ⟹  (M = 16)) ∧
    (R_A_16 + 0 = 16) ∧
    (R_A_16 = 16))
   [‘ADDI R_A_16,R_A,0‘]
   (((sigD = EMPTY_BAG) ∧
     (∀M. M IN_B sigD ⟹  (M = 16)) ∧
     (R_A = 16) ∧
     (R_A_16 = 16)) ∧
    (NZ = ¬(R_A_16 + 0 = 0)) ∧
    (GE = (R_A_16 + 0) ≥ 0))
    Run time: 0.5s
    Intermediate theorems generated: 18
```

The next step in this proof is to eliminate the (unnecessary) conditions on the flags NZ and GE from the postcondition. This step was performed simply for cosmetic reasons only, and is displayed in HOL session box 3. The help function `help_r_conseq` is applied to `th0` and the desired postcondition $\{\sigma_D = \Phi \ \wedge \ Inv \ \wedge \ R\_A = 16\}$. `help_r_conseq` constructs the implication required by the right consequence rule, and sets it as an HOL proof goal.

This goal is then easily demonstrated.

```
help_r_conseq th0                                                    3
      "(sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧
       (R_A = 16)";;

   "((sigD = EMPTY_BAG) ∧
     (∀M. M IN_B sigD ⟹  (M = 16)) ∧
     (R_A = 16) ∧
     (R_A_16 = 16)) ∧
    (NZ = ¬(R_A_16 + 0 = 0)) ∧
    (GE = (R_A_16 + 0) ≥ 0) ⟹
    (sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A = 16)"

   () : void
   Run time: 2.4s

#e(DISCH_TAC THEN ASM_REWRITE_TAC []);;
   OK..
   goal proved
   ⊢ ((sigD = EMPTY_BAG) ∧
       (∀M. M IN_B sigD ⟹  (M = 16)) ∧
       (R_A = 16) ∧
       (R_A_16 = 16)) ∧
      (NZ = ¬(R_A_16 + 0 = 0)) ∧
      (GE = (R_A_16 + 0) ≥ 0) ⟹
      (sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A = 16)

   Previous subproof:
   goal proved
   () : void
   Run time: 2.8s
   Intermediate theorems generated: 54
```

In HOL session box 4, the right consequence rule is applied to `th0` and the implication demonstrated in the previous session box (shown here as `top_thm()`). The resulting triple is displayed in Figure 7.3.

$$\{\sigma_D = \Phi \ \wedge \ Inv \ \wedge \ R\_A\_16 + 0 = 16 \ \wedge \ R\_A\_16 = 16\}$$
$$\texttt{ADDI} \ \ R^A_{16}, \ R\_A, \ 0;$$
$$\{\sigma_D = \Phi \ \wedge \ Inv \ \wedge \ R\_A = 16\}$$

Figure 7.3: Proof Outline for HOL Session Box 4

```
#let th0 = r_conseq th0 (top_thm());;                                                    4

   th0 =
   . ⊢ Prov
        ((sigD = EMPTY_BAG) ∧
         (∀M. M IN_B sigD ⟹  (M = 16)) ∧
         (R_A_16 + 0 = 16) ∧
         (R_A_16 = 16))
        [‘ADDI R_A_16,R_A,0‘]
        ((sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A = 16))
   Run time: 0.1s
   Intermediate theorems generated: 10
```

In HOL session box 5, the axiom schema for **send** is applied, returning the theorem (called th0) pictured in Figure 7.4. The **send** command assigns the value of $R\_A\_16 + 0$ to R_A and sets the condition flags NZ and GE appropriately. This command is, essentially, an assignment of $R\_A\_16 + 0$ to R_A.

$\{16 \in (\sigma_D \oplus R\_A) \wedge$
$\sigma_D \oplus R\_A = \sigma_D \oplus 16 \wedge$
$Inv \wedge R\_A = 16\}$

send R_A *to* D

$\{16 \in \sigma_D \wedge$
$\sigma_D = \Phi \oplus 16 \wedge$
$Inv \wedge R\_A = 16\}$

Figure 7.4: Proof Outline for HOL Session Box 5

```
#let th1 = mk_send ``[` send R_A to D `]"                                    5
"(16 IN_B sigD) ∧ (sigD = (16 INSERT_B EMPTY_BAG)) ∧
(∀M:num. (M IN_B sigD) ⟹  (M = 16)) ∧ (R_A = 16)"
                              "R_A" "sigD";;

   th1 =
   ⊢ Prov
      (16 IN_B (R_A INSERT_B sigD) ∧
       (R_A INSERT_B sigD = 16 INSERT_B EMPTY_BAG) ∧
       (∀M. M IN_B (R_A INSERT_B sigD) ⟹  (M = 16)) ∧
       (R_A = 16))
      [` send R_A to D `]
      (16 IN_B sigD ∧
       (sigD = 16 INSERT_B EMPTY_BAG) ∧
       (∀M. M IN_B sigD ⟹  (M = 16)) ∧
       (R_A = 16))
   Run time: 0.3s
   Intermediate theorems generated: 25
```

6

```
#set_goal([],
"((sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. (M IN_B sigD) ⟹  (M=16))) ⟹
             ((16 IN_B ( R_A INSERT_B sigD)) ∧
              ((R_A INSERT_B sigD) = (16 INSERT_B EMPTY_BAG)) ∧
              (∀M. (M IN_B (R_A INSERT_B sigD)) ⟹  (M = 16)) ∧
              (R_A = 16))");;

   "(sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ⟹
    16 IN_B (R_A INSERT_B sigD) ∧
    (R_A INSERT_B sigD = 16 INSERT_B EMPTY_BAG) ∧
    (∀M. M IN_B (R_A INSERT_B sigD) ⟹  (M = 16)) ∧
    (R_A = 16)"

   () : void
   Run time: 0.3s

#e(DISCH_TAC THEN
CONJ_TAC THEN
REWRITE_TAC [IN_B]);;

   Theorem IN_B autoloaded from theory `bags`.
   IN_B =
   ⊢ (∀x. x IN_B EMPTY_BAG = F) ∧
      (∀x y b. x IN_B (y INSERT_B b) = (x = y) ∨ x IN_B b)
   Run time: 2.4s

   OK..
   2 subgoals
   Current subgoal:
   "(16 = R_A) ∨ 16 IN_B sigD"
       [ "(sigD = EMPTY_BAG) ∧
          (R_A = 16) ∧
          (∀M. M IN_B sigD ⟹  (M = 16))" ]

   () : void
   Run time: 2.7s
   Intermediate theorems generated: 41
```

```
#e(ASM_REWRITE_TAC [(DISCH "R_A=16" (ASSUME "16 = R_A"))] );;                    7

   OK..
   goal proved
   . ⊢ (16 = R_A) ∨ 16 IN_B sigD

   Previous subproof:
   "(R_A INSERT_B sigD = 16 INSERT_B EMPTY_BAG) ∧
    (∀M. (M = R_A) ∨ M IN_B sigD ⟹  (M = 16)) ∧
    (R_A = 16)"
       [ "(sigD = EMPTY_BAG) ∧
          (R_A = 16) ∧
          (∀M. M IN_B sigD ⟹  (M = 16))" ]

   () : void
   Run time: 2.6s
   Intermediate theorems generated: 26
```

8

```
#e(CONJ_TAC THEN
ASM_REWRITE_TAC [(DISCH "R_A=16" (ASSUME "16 = R_A"))] THEN
ASM_REWRITE_TAC [] THEN
REWRITE_TAC [IN_B]);;

   OK..
   goal proved
   . ⊢ (R_A INSERT_B sigD = 16 INSERT_B EMPTY_BAG) ∧
        (∀M. (M = R_A) ∨ M IN_B sigD ⟹  (M = 16)) ∧
        (R_A = 16)
   ⊢ (sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ⟹
      16 IN_B (R_A INSERT_B sigD) ∧
      (R_A INSERT_B sigD = 16 INSERT_B EMPTY_BAG) ∧
      (∀M. M IN_B (R_A INSERT_B sigD) ⟹  (M = 16)) ∧
      (R_A = 16)

   Previous subproof:
   goal proved
   () : void
   Run time: 5.4s
   Intermediate theorems generated: 121

#let help_thm0 = top_thm();;

   help_thm0 =
   ⊢ (sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ⟹
      16 IN_B (R_A INSERT_B sigD) ∧
      (R_A INSERT_B sigD = 16 INSERT_B EMPTY_BAG) ∧
      (∀M. M IN_B (R_A INSERT_B sigD) ⟹  (M = 16)) ∧
      (R_A = 16)
   Run time: 0.0s
```

9

```
#let th2 = l_conseq th1 help_thm0;;

   th2 =
   . ⊢ Prov
        ((sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16)))
        [‘ send R_A to D ‘]
        (16 IN_B sigD ∧
         (sigD = 16 INSERT_B EMPTY_BAG) ∧
         (∀M. M IN_B sigD ⟹  (M = 16)) ∧
         (R_A = 16))
   Run time: 0.0s
   Intermediate theorems generated: 10
```

$$\{\sigma_D = \Phi \wedge$$
$$R\_A = 16 \wedge Inv\}$$
$$\textbf{send } R\_A \textit{ to } D$$
$$\{16 \in \sigma_D \wedge$$
$$\sigma_D = \Phi \oplus 16 \wedge$$
$$Inv \wedge R\_A = 16\}$$

Figure 7.5: Proof Outline for HOL Session Box 9

```
help_compose th0 th2;;                                                    10

   "(sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A = 16) ⟹
    (sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))"

   () : void
   Run time: 0.0s

#e(DISCH_TAC THEN ASM_REWRITE_TAC []);;

   OK..
   goal proved
   ⊢ (sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A = 16) ⟹
      (sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))

   Previous subproof:
   goal proved
   () : void
   Run time: 2.7s
   Intermediate theorems generated: 48

#let comp_thm0 = top_thm();;

   comp_thm0 =
   ⊢ (sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A = 16) ⟹
      (sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))
   Run time: 0.0s
```

$\{\sigma_D = \Phi \ \wedge \ Inv \wedge$
$\quad R\_A\_16 + 0 = 16 \ \wedge \ R\_A\_16 = 16\}$
ADDI R_A_16, R_A, 0
send R_A *to* D
$\{16 \in \ \sigma_D \wedge$
$\quad \sigma_D \ = \ \Phi \oplus 16 \wedge$
$\quad Inv \ \wedge \ R\_A = 16 \ \}$

Figure 7.6: Proof Outline for HOL Session Box 11

```
#let th3 = compose th0 th2 comp_thm0;;                                    11

   th3 =
   ... ⊢ Prov
         ((sigD = EMPTY_BAG) ∧
          (∀M. M IN_B sigD ⟹  (M = 16)) ∧
          (R_A_16 + 0 = 16) ∧
          (R_A_16 = 16))
         [‘ADDI R_A_16,R_A,0‘;‘ send R_A to D ‘]
         (16 IN_B sigD ∧
          (sigD = 16 INSERT_B EMPTY_BAG) ∧
          (∀M. M IN_B sigD ⟹  (M = 16)) ∧
          (R_A = 16))
   Run time: 0.5s
   Intermediate theorems generated: 43
```

12

```
#help_l_conseq th3 "((sigD = EMPTY_BAG) ∧
                    (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A_16 = 16))";;

   "(sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A_16 = 16) ⟹
    (sigD = EMPTY_BAG) ∧
    (∀M. M IN_B sigD ⟹  (M = 16)) ∧
    (R_A_16 + 0 = 16) ∧
    (R_A_16 = 16)"

   () : void
   Run time: 2.5s

#e(DISCH_TAC THEN ASM_REWRITE_TAC [(SPEC "16" ADD_0)]);;

   Theorem ADD_0 autoloaded from theory `arithmetic`.
   ADD_0 = ⊢ ∀m. m + 0 = m
   Run time: 2.4s

   OK..
   goal proved
   ⊢ (sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A_16 = 16) ⟹
      (sigD = EMPTY_BAG) ∧
      (∀M. M IN_B sigD ⟹  (M = 16)) ∧
      (R_A_16 + 0 = 16) ∧
      (R_A_16 = 16)

   Previous subproof:
   goal proved
   () : void
   Run time: 0.6s
   Intermediate theorems generated: 61
```

$\{\sigma_D = \Phi \ \wedge \ Inv \wedge$
$\quad R\_A\_16 = 16\}$
ADDI R_A_16, R_A, 0 ;
send R_A *to* D
$\{16 \in \ \sigma_D \wedge$
$\quad \sigma_D \ = \ \Phi \oplus 16 \wedge$
$\quad Inv \ \wedge \ R\_A = 16 \}$

Figure 7.7: Proof Outline for HOL Session Box 13

13

```
#let th4 = l_conseq th3 (top_thm());;

   th4 =
   .... ⊢ Prov
           ((sigD = EMPTY_BAG) ∧
            (∀M. M IN_B sigD ⟹  (M = 16)) ∧
            (R_A_16 = 16))
           [‘ADDI R_A_16,R_A,0‘;‘ send R_A to D ‘]
           (16 IN_B sigD ∧
            (sigD = 16 INSERT_B EMPTY_BAG) ∧
            (∀M. M IN_B sigD ⟹  (M = 16)) ∧
            (R_A = 16))
   Run time: 0.1s
   Intermediate theorems generated: 10
```

14

```
#mk_Sat_rec
   "(rhoD = EMPTY_BAG) ∧ (∀M:num. (M IN_B sigD) ⟹  (M = 16))"
   "(R_D=16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))"
   "R_D" "MTEXT" "sigD:(num)bag" "rhoD:(num)bag";;


   "((rhoD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16))) ∧
    MTEXT IN_B (sigD DIFF_B rhoD) ⟹
    (MTEXT = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))"
```

```
#e(STRIP_TAC THEN ASM_REWRITE_TAC []);;                                    15

   OK..
   "MTEXT = 16"
       [ "rhoD = EMPTY_BAG" ]
       [ "∀M. M IN_B sigD ⟹  (M = 16)" ]
       [ "MTEXT IN_B (sigD DIFF_B rhoD)" ]

#DIFF_B_LEMMA;;

   ⊢ ∀x b c. x IN_B (b DIFF_B c) ⟹  x IN_B b
   Run time: 0.0s

#e(ASSUM_LIST (λthl. ASSUME_TAC
                (REWRITE_RULE [(el 1 thl)]
                 (SPEC "rhoD" (SPEC "sigD" (SPEC "MTEXT"
                  (INST_TYPE [":num",":*"] DIFF_B_LEMMA)))))));;

   OK..
   "MTEXT = 16"
       [ "rhoD = EMPTY_BAG" ]
       [ "∀M. M IN_B sigD ⟹  (M = 16)" ]
       [ "MTEXT IN_B (sigD DIFF_B rhoD)" ]
       [ "MTEXT IN_B sigD" ]

#e(ASSUM_LIST(λthl. ASM_REWRITE_TAC
             [(MP (SPEC "MTEXT" (el 3 thl)) (el 1 thl))]));;

   OK..
   goal proved
   .. ⊢ MTEXT = 16
   .. ⊢ MTEXT = 16
   ⊢ ((rhoD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16))) ∧
      MTEXT IN_B (sigD DIFF_B rhoD) ⟹
      (MTEXT = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))
```

$\{\rho_D = \Phi\}$

receive R_D

$\{R\_D = 16\}$

Figure 7.8: Proof Outline for HOL Session Box 16

```
#let sat_r = top_thm();;                                                    16


   sat_r =
   ⊢ ((rhoD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16))) ∧
     MTEXT IN_B (sigD DIFF_B rhoD) ⟹
      (MTEXT = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))
   Run time: 0.0s

#let th0_D = mk_receive "[‘ receive R_D ‘]" sat_r "R_D"
         "(R_D=16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))";;

   th0_D =
   ⊢ Prov
      ((rhoD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)))
      [‘ receive R_D ‘]
      ((R_D = 16) ∧ (∀M. M IN_B sigD ⟹  (M = 16)))
   Run time: 0.1s
   Intermediate theorems generated: 22
```

7.2 A More Complicated Example

Figure 7.9 contains a proof outline for the second example. There are four processes—A, B, C, and D. Processes A, B, and C each send one message to process D. Process D has a **while** loop which receives messages. If this loop halts, then the register R_4 is shown to contain the maximum value received by D. Note that the loop halting depends on whether $R_3 = 0$ initially. The instruction `SUB` was included to give an example use of the complete language. This particular application is superfluous—the result is written to register R_5 which is never used anywhere else in the code. We present the derivation of the Hoare triple

in Figure 7.9 as an example of our system, rather than as exemplary code verification.

In HOL session box 1, the requisite constants are defined for the example in Figure 7.9. Registers R_0 through R_5 are used by process D exclusively, so no process context was indicated (e.g., as with *R_0_D*). Message variables — *mess*, *mess_A*, *mess_B*, *mess_C* — are given. Processes A, B, and C send *mess*, *mess_A*, *mess_B*, and *mess_C*, respectively, to process D. The constant *mess* is used in the satisfaction formula for the receive statement of process D (as with *MTEXT* in the previous example). Finally, the send and receive bags—σ_D and ρ_D — are represented by `sigD` and `rhoD` as in the previous example.

1

```
new_constant(‘R0‘,":num");;

new_constant(‘R1‘,":num");;

new_constant(‘R2‘,":num");;

new_constant(‘R3‘,":num");;

new_constant(‘R4‘,":num");;

new_constant(‘R5‘,":num");;

new_constant(‘mess‘,":num");;

new_constant(‘mess_A‘,":num");;

new_constant(‘mess_B‘,":num");;

new_constant(‘mess_C‘,":num");;

let sigD_DEF = new_definition(‘sigD_DEF‘,
                "sigD = (ABS_bag λx:num. 0)");;

let rhoD_DEF = new_definition(‘rhoD_DEF‘,
                "rhoD = (ABS_bag λx:num. 0)");;
```

In HOL session box 2, we define the system invariant *Inv* and the postcondition of the **receive** statement Q. We strengthened the invariant and postcondition to *Inv'* and *Q'* by adding conjunct $R_3 \neq 3$ to both *Inv* and Q. *Inv*, *Inv'*, Q, and *Q'* are ML constants—we use

A ::

$\{\ Inv\ \}$ **send** $mess_A$ **to** D $\{\ Inv\ \}$

B ::

$\{\ Inv\ \}$ **send** $mess_B$ **to** D $\{\ Inv\ \}$

C ::

$\{\ Inv\ \}$ **send** $mess_C$ **to** D $\{\ Inv\ \}$

D ::

$\{\ Inv\ \}$

while $R_3 \neq 3$ **do**

receive R_2

if $R_3 = 0$ **then**

`ADDI` $R_2,\ R_4,\ 0$

endif

`SUB` $R_2,\ R_4,\ R_5$

if $R_3 \neq 0$ **then**

if $R_4 < R_2$ **then**

`ADDI` $R_2,\ R_4,\ 0$

endif

endif

`ADDI` $R_3,\ R_3,\ 1$

endwhile

$\{\ (\forall M\ (M \in \rho_D) \implies (M \leq R_4)) \wedge$
$(R_4 \in \rho_D)\ \}$

Inv ::

$\{\ (\forall M\ (M \in \rho_D) \implies (M \leq R_4)) \wedge$
$(0 \leq R_3) \wedge$
$(R_3 = 0 \implies \rho_D = \Phi) \wedge$
$(R_3 \neq 0 \implies R_4 \in \rho_D)\ \}$

Figure 7.9: A More Complicated Example

them as *abbreviations* alone. One could simply type the actual invariant whenever required, rather than giving them names, but this would be time-consuming and error-prone.

2

```
#let Inv = "(∀M. (M IN_B rhoD) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧
           ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹  (R4 IN_B rhoD))";;

#let Q = "(∀M. (M IN_B (rhoD SUB_B R2)) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧ (R2 IN_B rhoD) ∧
           ((R3 = 0) ⟹  ((rhoD SUB_B R2) = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹  (R4 IN_B (rhoD SUB_B R2)))";;

#let Inv' = mk_conj(Inv,"¬(R3 = 3)");;

#let Q' = mk_conj(Q,"¬(R3 = 3)");;
```

In HOL session boxes 3 and 4, we prove the theorem displayed in Figure 7.10. `mk_Sat_rec` is applied to the appropriate variables, and `mk_receive` returns the appropriate theorem. Note that the details of proving the satisfaction in session box 3 have been left out of this presentation.

3

```
#mk_Sat_rec
Inv'
Q'
"R2" "mess" "sigD:(num)bag" "rhoD:(num)bag";;

"(((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
   0 ≤ R3 ∧
   ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
   (¬(R3 = 0) ⟹  R4 IN_B rhoD)) ∧
  ¬(R3 = 3)) ∧
 mess IN_B (sigD DIFF_B rhoD) ⟹
 ((∀M. M IN_B ((mess INSERT_B rhoD) SUB_B mess) ⟹  M ≤ R4) ∧
  0 ≤ R3 ∧
  mess IN_B (mess INSERT_B rhoD) ∧
  ((R3 = 0) ⟹  ((mess INSERT_B rhoD) SUB_B mess = EMPTY_BAG)) ∧
  (¬(R3 = 0) ⟹  R4 IN_B ((mess INSERT_B rhoD) SUB_B mess))) ∧
 ¬(R3 = 3)"
```

$\{Inv'\}$

receive R_2

$\{Q'\}$

Figure 7.10: Proof Outline for HOL Session Box 4

4

```
 proof deleted ...

#let sat_thm = top_thm();;

#let th1 = mk_receive "[‘ receive R2 ‘]" sat_thm "R2" Q';;
th1 =
⊢ Prov
   (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
     0 ≤ R3 ∧
     ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
     (¬(R3 = 0) ⟹  R4 IN_B rhoD)) ∧
    ¬(R3 = 3))
   [‘ receive R2 ‘]
   (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
     0 ≤ R3 ∧
     R2 IN_B rhoD ∧
     ((R3 = 0) ⟹  (rhoD SUB_B R2 = EMPTY_BAG)) ∧
     (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2))) ∧
    ¬(R3 = 3))
```

The triple in Figure 7.11 is demonstrated in HOL session box 5. `mk_ADDI` is applied to `new_post`, with target register "R_4", source register "R_2", and immediate value "0". The resulting theorem is named `th2`. This `ADDI` instruction occurs in the first **if** statement of process D.

$$
\begin{array}{l}
\{(\forall\, M.\, M \in \rho_D \ominus R_2 \implies M \leq R_2 + 0) \wedge \\
\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \ \wedge \\
\quad (R_3 = 0 \implies (R_2 = R_2 + 0 \ \wedge\ (R_2 + 0) \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi)) \wedge \\
\quad (R_3 \neq 0 \Longrightarrow (R_2 + 0) \in (\rho_D \ominus R_2)) \ \wedge\ R_3 \neq 3\} \\
\qquad\qquad \texttt{ADDI}\ \ R_2,\ R_4,\ 0 \\
\qquad\qquad\quad \{(\forall\, M.\, M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge \\
\qquad\qquad\quad\ 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \ \wedge \\
\qquad\qquad\quad\ (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi) \wedge \\
\qquad\qquad\quad\ (R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \wedge \\
\qquad\qquad\quad\ R_3 \neq 3 \ \wedge\ NZ = (R_2 + 0 = 0) \ \wedge\ GE = (R_2 + 0) \geq 0)\}
\end{array}
$$

Figure 7.11: Proof Outline for HOL Session Box 5

```
let new_post = "(∀M. (M IN_B (rhoD SUB_B R2)) ⟹  (M ≤ R4)) ∧                    5
        (0 ≤ R3) ∧ (R2 IN_B rhoD) ∧
        ((R3 = 0) ⟹
        ((R2 = R4) ∧ (R4 IN_B rhoD) ∧ ((rhoD SUB_B R2) = EMPTY_BAG))) ∧
        (¬(R3 = 0) ⟹  (R4 IN_B (rhoD SUB_B R2))) ∧ ¬(R3 = 3)";;

let th2 = mk_ADDI "[‘ADDI R2,R4,0‘]" new_post "R4" "R2" "0";;

th2 =
⊢ Prov
   ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ (R2 + 0)) ∧
    0 ≤ R3 ∧
    R2 IN_B rhoD ∧
    ((R3 = 0) ⟹
     (R2 = R2 + 0) ∧ (R2 + 0) IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
    (¬(R3 = 0) ⟹  (R2 + 0) IN_B (rhoD SUB_B R2)) ∧
    ¬(R3 = 3))
   [‘ADDI R2,R4,0‘]
   (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
     0 ≤ R3 ∧
     R2 IN_B rhoD ∧
     ((R3 = 0) ⟹
      (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
     (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
     ¬(R3 = 3)) ∧
    (NZ = ¬(R2 + 0 = 0)) ∧
    (GE = (R2 + 0) ≥ 0))
```

$$
\begin{array}{l}
\{(\forall\, M.\, M \in \rho_D \ominus R_2 \implies M \leq R_2 + 0) \wedge \\
\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \wedge \\
\quad (R_3 = 0 \implies (R_2 = R_2 + 0 \ \wedge\ (R_2 + 0) \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi)) \\
\quad (R_3 \neq 0 \Longrightarrow (R_2 + 0) \in (\rho_D \ominus R_2)) \wedge \\
\quad R_3 \neq 3 \ \wedge\ R_3 = 0\} \\
\qquad\qquad \texttt{ADDI}\ R_2,\ R_4,\ 0 \\
\qquad\qquad\quad \{(\forall\, M.M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge \\
\qquad\qquad\qquad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \wedge \\
\qquad\qquad\qquad ((R_3 = 0) \implies (R_2 = R_4) \ \wedge\ R_4 \in \rho_D \ \wedge\ (\rho_D \ominus R_2 = \Phi)) \wedge \\
\qquad\qquad\qquad (R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \wedge \\
\qquad\qquad\qquad R_3 \neq 3 \ \wedge\ NZ = (R_2 + 0 = 0) \ \wedge\ GE = (R_2 + 0 \geq 0)\}
\end{array}
$$

Figure 7.12: Proof Outline for HOL Session Box 6

Figure 7.12 contains the proof outline which is demonstrated in HOL session box 6. In HOL session boxes 6 and 7, the theorem necessary for applying the IF inference rule is derived.

6

```
#let Q_if1 = mk_conj(Q',"R3 = 0");;
Q_if1 =
"(((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
   0 ≤ R3 ∧
   R2 IN_B rhoD ∧
   ((R3 = 0) ⟹  (rhoD SUB_B R2 = EMPTY_BAG)) ∧
   (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2))) ∧
  ¬(R3 = 3)) ∧
 (R3 = 0)"

#help_l_conseq th2 Q_if1;;
 proof deleted ...

#let th2 = l_conseq th2 (top_thm());;
th2 =
. ⊢ Prov
     ((((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
        0 ≤ R3 ∧
        R2 IN_B rhoD ∧
        ((R3 = 0) ⟹  (rhoD SUB_B R2 = EMPTY_BAG)) ∧
        (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2))) ∧
       ¬(R3 = 3)) ∧
      (R3 = 0))
     [‘ADDI R2,R4,0‘]
     (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
       0 ≤ R3 ∧
       R2 IN_B rhoD ∧
       ((R3 = 0) ⟹
        (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
       (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
       ¬(R3 = 3)) ∧
      (NZ = ¬(R2 + 0 = 0)) ∧
      (GE = (R2 + 0) ≥ 0))
```

$$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$$
$$0 \leq R_3 \ \wedge\ R_2 \in \rho_D \wedge$$
$$(R_3 = 0 \implies (\rho_D \ominus R_2 = \Phi)) \wedge$$
$$(R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2))) \wedge$$
$$R_3 \neq 3 \ \wedge\ R_3 = 0\}$$

$$\texttt{ADDI}\ R_2,\ R_4,\ 0$$

$$\{(\forall\, M.M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$$
$$0 \leq R_3 \ \wedge\ R_2 \in \rho_D \wedge$$
$$(R_3 = 0 \implies R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi) \wedge$$
$$(R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \ \wedge\ R_3 \neq 3)\}$$

Figure 7.13: Proof Outline for HOL Session Box 7

7

```
#help_r_conseq th2 new_post;;
 proof deleted ...

#let th2 = r_conseq th2 (top_thm());;
th2 =
.. ⊢ Prov
      ((((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
         0 ≤ R3 ∧
         R2 IN_B rhoD ∧
         ((R3 = 0) ⟹  (rhoD SUB_B R2 = EMPTY_BAG)) ∧
         (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2))) ∧
        ¬(R3 = 3)) ∧
       (R3 = 0))
      [‘ADDI R2,R4,0‘]
      ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
       0 ≤ R3 ∧
       R2 IN_B rhoD ∧
       ((R3 = 0) ⟹
        (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
       (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
       ¬(R3 = 3))
```

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \wedge$
$\quad (R_3 = 0 \implies (\rho_D \ominus R_2 = \Phi)) \wedge$
$\quad (R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \wedge$
$\quad R_3 \neq 3\}$

if $R_3 \neq 0$ **then**

`ADDI` $R_2,\ R_4,\ 0$

endif

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \wedge$
$\quad (R_3 = 0 \implies R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi) \wedge$
$\quad (R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \ \wedge\ R_3 \neq 3\}$

Figure 7.14: Proof Outline for HOL Session Box 8

Note that the theorem shown in HOL session box 7 can be abbreviated by:

$\{new\_post \wedge R_3 = 0\}$

`ADDI` $R_2, R_4, 0$

$\{new\_post\}$

The IF inference rule may then be applied, as it is in session box 8. The resulting theorem is displayed in Figure 7.14.

```
#let Q_ifnot1 = mk_conj(Q',"¬(R3 = 0)");;                                    8
Q_ifnot1 =
"(((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
   0 ≤ R3 ∧
   R2 IN_B rhoD ∧
   ((R3 = 0) ⟹  (rhoD SUB_B R2 = EMPTY_BAG)) ∧
   (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2))) ∧
  ¬(R3 = 3)) ∧
 ¬(R3 = 0)"

#set_goal([],"^Q_ifnot1 ⟹  ^new_post");;
 proof deleted ...
#let th2 = mk_if th2 (top_thm());;

th2 =
... ⊢ Prov
       (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
         0 ≤ R3 ∧
         R2 IN_B rhoD ∧
         ((R3 = 0) ⟹  (rhoD SUB_B R2 = EMPTY_BAG)) ∧
         (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2))) ∧
        ¬(R3 = 3))
       [‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘]
       ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
        0 ≤ R3 ∧
        R2 IN_B rhoD ∧
        ((R3 = 0) ⟹
         (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
        (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
        ¬(R3 = 3))
```

In HOL session box 9, theorems `th1` and `th2` are composed with the aid of help function `help_compose`. The proof outline for the resulting triple is displayed in Figure 7.15.

$\{Inv'\}$

receive R_2

if $R_3 \neq 0$ **then** ADDI R_2, R_4, 0 **endif**

$\{(\forall M.M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$

$0 \leq R_3 \wedge R_2 \in \rho_D \wedge$

$(R_3 = 0 \implies R_2 = R_4 \wedge R_4 \in \rho_D \wedge \rho_D \ominus R_2 = \Phi) \wedge$

$(R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \wedge R_3 \neq 3\}$

Figure 7.15: Proof Outline for HOL Session Box 9

9

```
#help_compose th1 th2;;
 proof deleted ...

#let th3 = compose th1 th2 (top_thm());;
th3 =
.... ⊢ Prov
        (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
          0 ≤ R3 ∧
          ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
          (¬(R3 = 0) ⟹  R4 IN_B rhoD)) ∧
         ¬(R3 = 3))
        [‘ receive R2 ‘;‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘]
        ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
         0 ≤ R3 ∧
         R2 IN_B rhoD ∧
         ((R3 = 0) ⟹
          (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
         (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
         ¬(R3 = 3))
```

Recall that the instruction SUB was included to give an example use of the complete language. This particular application is superfluous—the result is written to register R_5 which is never used anywhere else in the code. HOL session boxes 10 and 11 display the application of mk_SUB and compose. The resulting theorem is displayed in Figure 7.16 and is named "th4".

$$
\begin{array}{l}
\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge \\
\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \ \wedge \\
\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi)) \wedge \\
\quad (R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \wedge \\
\quad R_3 \neq 3\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{SUB}\ R_2,\ R_4,\ R_5 \\
\quad \{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge \\
\qquad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \ \wedge \\
\qquad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi)) \wedge \\
\qquad (R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \wedge \\
\qquad R_3 \neq 3 \wedge \\
\qquad (NZ = (R_4 \neq R_2)) \ \wedge\ (GE = R_4 \geq R_2)\}
\end{array}
$$

Figure 7.16: Proof Outline for HOL Session Box 10

10

```
#let th4 = mk_SUB "[‘SUB R2,R4,R5‘]" new_post "R2" "R4" "R5";;
th4 =
⊢ Prov
   ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
    0 ≤ R3 ∧
    R2 IN_B rhoD ∧
    ((R3 = 0) ⟹
     (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
    (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
    ¬(R3 = 3))
   [‘SUB R2,R4,R5‘]
   (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
     0 ≤ R3 ∧
     R2 IN_B rhoD ∧
     ((R3 = 0) ⟹
      (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
     (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
     ¬(R3 = 3)) ∧
    (NZ = ¬(R4 = R2)) ∧
    (GE = R4 ≥ R2))
```

11

```
#let th4 = compose th3 th4 (top_thm());;

th4 =
 ⊢ Prov
   (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
      0 ≤ R3 ∧
      ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
      (¬(R3 = 0) ⟹  R4 IN_B rhoD)) ∧
      ¬(R3 = 3))
   [‘ receive R2 ‘;‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘;‘SUB R2,R4,R5‘]
    (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
       0 ≤ R3 ∧
       R2 IN_B rhoD ∧
    ((R3 = 0) ⟹
      (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
      (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
      ¬(R3 = 3)) ∧
      (NZ = ¬(R4 = R2)) ∧
      (GE = R4 ≥ R2))
```

$\{Inv'\}$

receive R_2

if $R_3 \neq 0$ **then**

  `ADDI` $R_2,\ R_4,\ 0$

endif

`SUB` $R_2,\ R_4,\ R_5$

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$

$0 \leq R_3 \ \wedge\ R_2 \in \rho_D \ \wedge$

$(R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge\ \rho_D \ominus R_2 = \Phi)) \wedge$

$(R_3 \neq 0 \implies R_4 \in (\rho_D \ominus R_2)) \wedge$

$R_3 \neq 3 \wedge$

$(NZ = (R_4 \neq R_2)) \ \wedge\ (GE = R_4 \geq R_2)\}$

Figure 7.17: Proof Outline for HOL Session Box 11

$$
\begin{array}{l}
\{(\forall\, M.\ M \in \rho_D \implies (M \leq R_2 + 0)) \wedge \\
\quad 0 \leq R_3 \wedge \\
\quad (R_3 = 0 \implies \rho_D = \Phi) \wedge \\
\quad (R_3 \neq 0 \implies (R_2 + 0) \in \rho_D) \wedge \\
\quad R_2 + 0 \in \rho_D \ \wedge\ R_3 \neq 3\} \\
\qquad\qquad \texttt{ADDI}\ \ R_2,\ R_4,\ 0 \\
\qquad\qquad\quad \{(\forall\, M.\ M \in \rho_D \implies (M \leq R_4)) \wedge \\
\qquad\qquad\qquad 0 \leq R_3 \wedge \\
\qquad\qquad\qquad (R_3 = 0 \implies \rho_D = \Phi) \wedge \\
\qquad\qquad\qquad (R_3 \neq 0 \implies R_4 \in \rho_D) \wedge \\
\qquad\qquad\qquad R_4 \in \rho D \ \wedge\ R_3 \neq 3 \wedge \\
\qquad\qquad\qquad (NZ = (R_2 + 0 \neq 0)) \ \wedge\ (GE = R_2 + 0 \geq 0)\}
\end{array}
$$

Figure 7.18: Proof Outline for HOL Session Box 12

We begin developing the proof outline for the second "nested" **if** statement in the code for process D in HOL session box 12. The proof outline itself is displayed in Figure 7.18. In session boxes 12 through 15, proofs are given of

$$\{Q\_outerif \wedge R_4 < R_2\}$$
$$\texttt{ADDI}\ \ R_2, R_4, 0$$
$$\{Inv\_spec\}$$

and

$$(Q\_outerif \wedge \neg(R_4 < R_2) \rightarrow Inv\_spec$$

where $Q\_outerif$ is defined in session box 13. This allows the inner **if** statement to be derived.

12

```
#let Inv_spec = "(∀M. (M IN_B rhoD) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧
           ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹  (R4 IN_B rhoD)) ∧
           (R4 IN_B rhoD) ∧ ¬(R3 = 3)";;

#let th5 = mk_ADDI "[‘ADDI R2,R4,0‘]" Inv_spec "R4" "R2" "0";;
th5 =
⊢ Prov
   ((∀M. M IN_B rhoD ⟹  M ≤ (R2 + 0)) ∧
    0 ≤ R3 ∧
    ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
    (¬(R3 = 0) ⟹  (R2 + 0) IN_B rhoD) ∧
    (R2 + 0) IN_B rhoD ∧
    ¬(R3 = 3))
   [‘ADDI R2,R4,0‘]
   (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
     0 ≤ R3 ∧
     ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
     (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
     R4 IN_B rhoD ∧
     ¬(R3 = 3)) ∧
    (NZ = ¬(R2 + 0 = 0)) ∧
    (GE = (R2 + 0) ≥ 0))
```

$$
\begin{array}{l}
\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge \\
\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho_D \ \wedge \\
\quad (R_3 = 0 \implies \rho_D = \Phi) \wedge \\
\quad (R_3 \neq 0 \implies R_4 \in \rho_D \ominus R_2) \wedge \\
\quad R_3 \neq 3 \ \wedge\ R_3 \neq 0 \ \wedge\ R_4 < R_2\} \\
\qquad\qquad \texttt{ADDI}\ R_2,\ R_4,\ 0 \\
\qquad\qquad\quad \{(\forall\, M.\ M \in \rho_D \implies (M \leq R_4)) \wedge \\
\qquad\qquad\qquad 0 \leq R_3 \wedge \\
\qquad\qquad\qquad (R_3 = 0 \implies \rho_D = \Phi) \wedge \\
\qquad\qquad\qquad (R_3 \neq 0 \implies R_4 \in \rho_D) \wedge \\
\qquad\qquad\qquad R_4 \in \rho D \ \wedge\ R_3 \neq 3 \\
\qquad\qquad\qquad (NZ = (R_2 + 0 \neq 0)) \ \wedge\ (GE = R_2 + 0 \geq 0)\}
\end{array}
$$

Figure 7.19: Proof Outline for HOL Session Box 13

We demonstrate the proof outline in Figure 7.19 in session box 13. This theorem can be abbreviated as:

$$\{Q\_outerif \wedge R_4 < R_2\}$$
$$\texttt{ADDI}\ \ R_2, R_4, 0$$
$$\{Inv\_spec \wedge CF\}$$

where CF are the condition flag assignments.

```
#let Q_outerif = mk_conj(new_post,"¬(R3=0)");;                                  13
Q_outerif =
"((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
  0 ≤ R3 ∧
  R2 IN_B rhoD ∧
  ((R3 = 0) ⟹  (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
  (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
  ¬(R3 = 3)) ∧
 ¬(R3 = 0)"

#let Q_innerif = mk_conj(Q_outerif,"R4 < R2");;

#help_l_conseq th5 Q_innerif;;
 proof deleted ...

#let th5 = l_conseq th5 (top_thm());;
th5 =
. ⊢ Prov
     ((((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
        0 ≤ R3 ∧
        R2 IN_B rhoD ∧
        ((R3 = 0) ⟹
         (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
        (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
        ¬(R3 = 3)) ∧
       ¬(R3 = 0)) ∧
      R4 < R2)
     [‘ADDI R2,R4,0‘]
     (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
       0 ≤ R3 ∧
       ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
       (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
       R4 IN_B rhoD ∧
       ¬(R3 = 3)) ∧
      (NZ = ¬(R2 + 0 = 0)) ∧
      (GE = (R2 + 0) ≥ 0))
```

The theorem:

$$\{Q\_outerif \wedge R_4 < R_2\}$$
$$\texttt{ADDI}\ \ R_2, R_4, 0$$
$$\{Inv\_spec\}$$

$$
\begin{array}{l}
\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge \\
\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho D \\
\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge \rho_D \ominus R_2 = \Phi)) \wedge \\
\quad (R_3 \neq 0 \implies (R_4 \in \rho_D \ominus R_2) \wedge \\
\quad R_3 \neq 3 \ \wedge\ R_3 \neq 0\ \} \\
\qquad\qquad\qquad \texttt{ADDI}\ R_2,\ R_4,\ 0 \\
\qquad\qquad\quad \{(\forall\, M.\ M \in \rho_D \implies (M \leq R_4)) \wedge \\
\qquad\qquad\quad\ 0 \leq R_3 \wedge \\
\qquad\qquad\quad\ (R_3 = 0 \implies \rho_D = \Phi) \wedge \\
\qquad\qquad\quad\ (R_3 \neq 0 \implies R_4 \in \rho_D) \wedge \\
\qquad\qquad\quad\ R_4 \in \rho D \ \wedge\ R_3 \neq 3\}
\end{array}
$$

Figure 7.20: Proof Outline for HOL Session Box 14

is demonstrated in session box 14. The complete proof outline is shown in Figure 7.20.

14

```
#help_r_conseq th5 Inv_spec;;
 proof deleted ...

#let th5 = r_conseq th5 (top_thm());;
th5 =
.. ⊢ Prov
      ((((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
         0 ≤ R3 ∧
         R2 IN_B rhoD ∧
         ((R3 = 0) ⟹
          (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
         (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
         ¬(R3 = 3)) ∧
        ¬(R3 = 0)) ∧
       R4 < R2)
      [‘ADDI R2,R4,0‘]
      ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
       0 ≤ R3 ∧
       ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
       (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
       R4 IN_B rhoD ∧
       ¬(R3 = 3))
```

In session box 15, we demonstrate:

$$(Q\_outerif \wedge \neg(R_4 < R_2)) \Longrightarrow Inv\_spec$$

We may now apply the IF inference rule to the triple proved in session box 14. The resulting triple is displayed in Figure 7.21.

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho D$
$\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge \rho_D \ominus R_2 = \Phi)) \wedge$
$\quad (R_3 \neq 0 \implies (R_4 \in \rho_D \ominus R_2) \wedge$
$\quad R_3 \neq 3 \ \wedge\ R_3 \neq 0\ \}$

if $R_4 < R_2$ **then**

`ADDI` R_2, R_4, 0

endif

$\{(\forall\, M.\ M \in \rho_D \implies (M \leq R_4)) \wedge$
$\quad 0 \leq R_3 \wedge$
$\quad (R_3 = 0 \implies \rho_D = \Phi) \wedge$
$\quad (R_3 \neq 0 \implies R_4 \in \rho_D) \wedge$
$\quad R_4 \in \rho D \ \wedge\ R_3 \neq 3\}$

Figure 7.21: Proof Outline for HOL Session Box 15

15

```
#let Q_innernot = mk_conj(Q_outerif,"¬(R4 < R2)");;
Q_innernot =
"(((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
   0 ≤ R3 ∧
   R2 IN_B rhoD ∧
   ((R3 = 0) ⟹
    (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
   (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
   ¬(R3 = 3)) ∧
  ¬(R3 = 0)) ∧
 ¬R4 < R2"

#set_goal([], "^Q_innernot ⟹  ^Inv_spec");;
 proof deleted ...

#let th5 = mk_if th5 (top_thm());;
th5 =
... ⊢ Prov
       (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
         0 ≤ R3 ∧
         R2 IN_B rhoD ∧
         ((R3 = 0) ⟹
          (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
         (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
         ¬(R3 = 3)) ∧
        ¬(R3 = 0))
       [‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘]
       ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
        0 ≤ R3 ∧
        ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
        (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
        R4 IN_B rhoD ∧
        ¬(R3 = 3))
```

In session box 16, we weaken the postcondition of the theorem displayed in Figure 7.21 to Inv''. Inv'' is simply the result of deleting the conjunct "$R_3 = 0 \implies \rho_D = \Phi$" from the aforementioned postcondition. The resulting theorem is displayed in Figure 7.22.

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho D$
$\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge \rho_D \ominus R_2 = \Phi)) \wedge$
$\quad (R_3 \neq 0 \implies (R_4 \in \rho_D \ominus R_2) \wedge$
$\quad R_3 \neq 3 \ \wedge\ R_3 \neq 0\ \}$

if $R_4 < R_2$ **then**

ADDI R_2, R_4, 0

endif

$\{(\forall\, M.\ M \in \rho_D \implies (M \leq R_4)) \wedge$
$\quad 0 \leq R_3 \wedge$
$\quad (R_3 \neq 0 \implies R_4 \in \rho_D) \wedge$
$\quad R_4 \in \rho D \ \wedge\ R_3 \neq 3\}$

Figure 7.22: Proof Outline for HOL Session Box 16

16

```
#let Inv'' = "(∀M. (M IN_B rhoD) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧
           (¬(R3 = 0) ⟹  (R4 IN_B rhoD)) ∧
           (R4 IN_B rhoD) ∧
           ¬(R3 = 3)";;

#help_r_conseq th5 Inv'';;
 proof deleted ...

#let th5 = r_conseq th5 (top_thm());;
th5 =
.... ⊢ Prov
        (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
          0 ≤ R3 ∧
          R2 IN_B rhoD ∧
          ((R3 = 0) ⟹
           (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
          (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
          ¬(R3 = 3)) ∧
         ¬(R3 = 0))
        [‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘]
         ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
         0 ≤ R3 ∧
         (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
         R4 IN_B rhoD ∧
         ¬(R3 = 3))
```

The astute reader will notice that the triples proven in session boxes 16 and 17 are identical. Why bother to re-prove the same theorem? This serves as an example of why some kind of user-interface/proof manager (such as PM [12]) would greatly facilitate the use of this system. The HOL code for these two theorems had earlier in the development of this verification proven different theorems. However some time in the course of developing this proof, these two HOL code fragments became identical without my realizing it. The complexity of reading "raw" HOL code and of the verification's development conspired to keep me unaware of this redundancy.

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho D$
$\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge \rho_D \ominus R_2 = \Phi)) \wedge$
$\quad (R_3 \neq 0 \implies (R_4 \in \rho_D \ominus R_2) \wedge$
$\quad R_3 \neq 3 \ \wedge\ R_3 \neq 0\ \}$

if $R_4 < R_2$ **then**

ADDI R_2, R_4, 0

endif

$\{(\forall\, M.\ M \in \rho_D \implies (M \leq R_4)) \wedge$
$\quad 0 \leq R_3 \wedge$
$\quad (R_3 \neq 0 \implies R_4 \in \rho_D) \wedge$
$\quad R_4 \in \rho D \ \wedge\ R_3 \neq 3\}$

Figure 7.23: Proof Outline for HOL Session Box 17

17

```
#help_l_conseq th5 Q_outerif;;
 proof deleted ...

#let th5 = l_conseq th5 (top_thm());;
th5 =
..... ⊢ Prov
         (((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
           0 ≤ R3 ∧
           R2 IN_B rhoD ∧
           ((R3 = 0) ⟹
            (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
           ¬(R3 = 3)) ∧
          ¬(R3 = 0))
         [‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘]
         ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
          0 ≤ R3 ∧
          (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
          R4 IN_B rhoD ∧
          ¬(R3 = 3))
```

In session box 18, **mk_if** is applied, returning the theorem displayed in Figure 7.24.

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho D$
$\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge \rho_D \ominus R_2 = \Phi)) \wedge$
$\quad (R_3 \neq 0 \implies (R_4 \in \rho_D \ominus R_2)) \wedge$
$\quad R_3 \neq 3\}$

if $R_3 \neq 0$ **then**
  if $R_4 < R_2$ **then**
    `ADDI` R_2, R_4, 0
  endif
endif

$\{(\forall\, M.\ M \in \rho_D \implies (M \leq R_4)) \wedge$
$\quad 0 \leq R_3 \wedge$
$\quad (R_3 \neq 0 \implies R_4 \in \rho_D) \wedge$
$\quad R_4 \in \rho D \ \wedge\ R_3 \neq 3\}$

Figure 7.24: Proof Outline for HOL Session Box 18

18

```
#let Q_outernot = mk_conj(new_post,"¬¬(R3 = 0)");;
Q_outernot =
"((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
  0 ≤ R3 ∧
  R2 IN_B rhoD ∧
  ((R3 = 0) ⟹  (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
  (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
  ¬(R3 = 3)) ∧
 ¬¬(R3 = 0)"

set_goal([],"^Q_outernot ⟹  ^Inv'");;
 proof deleted ...

#let th5 = mk_if th5 (top_thm());;
th5 =
..... ⊢ Prov
          ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
           0 ≤ R3 ∧
           R2 IN_B rhoD ∧
           ((R3 = 0) ⟹
            (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
           ¬(R3 = 3))
          [` IF `;` IF `;`ADDI R2,R4,0`;` ENDIF `;` ENDIF `]
          ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
           0 ≤ R3 ∧
           (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
           R4 IN_B rhoD ∧
           ¬(R3 = 3))
```

In session box 19, the code for incrementing the R_3 register is introduced. Figure 7.25 displays the resulting theorem.

$$\{(\forall\, M.\, M \in \rho_D \implies M \leq R_4) \wedge$$
$$0 \leq R_3 + 1 \wedge$$
$$(R_3 + 1 \neq 0 \implies R_4 \in \rho_D) \wedge$$
$$R_3 + 1 \neq 0\}$$
$$\texttt{ADDI}\ R_3,\ R_3,\ 1$$
$$\{(\forall\, M.\, M \in \rho_D \implies M \leq R_4) \wedge$$
$$0 \leq R_3 \wedge$$
$$(R_3 \neq 0 \implies R_4 \in \rho_D) \wedge$$
$$R_3 \neq 0 \wedge$$
$$NZ = (R_3 + 1 \neq 0) \wedge GE = (R_3 + 1 \geq 0)\}$$

Figure 7.25: Proof Outline for HOL Session Box 19

19

```
#let Inv''' = "(∀M. (M IN_B rhoD) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧
           (¬(R3 = 0) ⟹  (R4 IN_B rhoD)) ∧
           ¬(R3 = 0)";;

#let th6 = mk_ADDI "[‘ADDI R3,R3,1‘]" Inv''' "R3" "R3" "1";;
th6 =
⊢ Prov
   ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
    0 ≤ (R3 + 1) ∧
    (¬(R3 + 1 = 0) ⟹  R4 IN_B rhoD) ∧
    ¬(R3 + 1 = 0))
   [‘ADDI R3,R3,1‘]
   (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
     0 ≤ R3 ∧
     (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
     ¬(R3 = 0)) ∧
    (NZ = ¬(R3 + 1 = 0)) ∧
    (GE = (R3 + 1) ≥ 0))
```

The proof outline for the nested **if** statement and the incrementing of R_3 are composed in

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho D$
$\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge \rho_D \ominus R_2 = \Phi)) \wedge$
$\quad (R_3 \neq 0 \implies (R_4 \in \rho_D \ominus R_2)) \wedge$
$\quad R_3 \neq 3\}$

if $R_3 \neq 0$ **then**
  if $R_4 < R_2$ **then**
    `ADDI` $R_2,\ R_4,\ 0$
  endif
endif
`ADDI` $R_3,\ R_3,\ 1$
  $\{(\forall\, M.\ M \in \rho_D \implies M \leq R_4) \wedge$
    $0 \leq R_3 \wedge$
    $(R_3 \neq 0 \implies R_4 \in \rho_D) \wedge$
    $R_3 \neq 0 \wedge$
    $NZ = (R_3 + 1 \neq 0) \ \wedge\ GE = (R_3 + 1 \geq 0)\}$

Figure 7.26: Proof Outline for HOL Session Box 20

session box 20. The resulting theorem is displayed in Figure 7.26 and is named "th6".

20

```
#help_compose th5 th6;;
 proof deleted ...

#let th6 = compose th5 th6 (top_thm());;
th6 =
....... ⊢ Prov
           ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
            0 ≤ R3 ∧
            R2 IN_B rhoD ∧
            ((R3 = 0) ⟹
             (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
            (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
            ¬(R3 = 3))
           [‘ IF ‘;‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘;‘ ENDIF ‘;
            ‘ADDI R3,R3,1‘]
           (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
             0 ≤ R3 ∧
             (¬(R3 = 0) ⟹  R4 IN_B rhoD) ∧
             ¬(R3 = 0)) ∧
            (NZ = ¬(R3 + 1 = 0)) ∧
            (GE = (R3 + 1) ≥ 0))
```

The postcondition of th6 is weakened in session box 21 to Inv, and the result is renamed "th6". The associated triple is displayed in Figure 7.27.

$\{(\forall\, M.\ M \in (\rho_D \ominus R_2) \implies M \leq R_4) \wedge$
$\quad 0 \leq R_3 \ \wedge\ R_2 \in \rho D$
$\quad (R_3 = 0 \implies (R_2 = R_4 \ \wedge\ R_4 \in \rho_D \ \wedge \rho_D \ominus R_2 = \Phi)) \wedge$
$\quad (R_3 \neq 0 \implies (R_4 \in \rho_D \ominus R_2)) \wedge$
$\quad R_3 \neq 3\}$

if $R_3 \neq 0$ **then**
  if $R_4 < R_2$ **then**
    `ADDI` R_2, R_4, 0
  endif
endif
`ADDI` R_3, R_3, 1
  $\{Inv\}$

Figure 7.27: Proof Outline for HOL Session Box 21

21

```
#help_r_conseq th6 Inv;;
 proof deleted ...

#let th6 = r_conseq th6 (top_thm());;
th6 =
........ ⊢ Prov
            ((∀M. M IN_B (rhoD SUB_B R2) ⟹  M ≤ R4) ∧
             0 ≤ R3 ∧
             R2 IN_B rhoD ∧
             ((R3 = 0) ⟹
              (R2 = R4) ∧ R4 IN_B rhoD ∧ (rhoD SUB_B R2 = EMPTY_BAG)) ∧
             (¬(R3 = 0) ⟹  R4 IN_B (rhoD SUB_B R2)) ∧
             ¬(R3 = 3))
            [‘ IF ‘;‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘;‘ ENDIF ‘;
             ‘ADDI R3,R3,1‘]
            ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
             0 ≤ R3 ∧
             ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
             (¬(R3 = 0) ⟹  R4 IN_B rhoD))
```

In session box 22, th4 and th6 are composed, resulting in the triple displayed in Figure 7.28 and named "loop_thm".

$\{Inv \wedge R_3 \neq 3\}$

receive R_2
if $R_3 \neq 0$ **then** ADDI R_2, R_4, 0 **endif**
SUB R_2, R_4, R_5
if $R_3 \neq 0$ **then**
  if $R_4 < R_2$ **then**
    ADDI R_2, R_4, 0
  endif
endif
ADDI R_3, R_3, 1
$\{Inv\}$

Figure 7.28: Proof Outline for HOL Session Box 22

22

```
#help_compose th4 th6;;
 proof deleted ...

#let loop_thm = compose th4 th6 (top_thm());;
loop_thm =
.............. ⊢ Prov
                 (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
                   0 ≤ R3 ∧
                   ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
                   (¬(R3 = 0) ⟹  R4 IN_B rhoD)) ∧
                  ¬(R3 = 3))
                 [‘ receive R2 ‘;‘ IF ‘;‘ADDI R2,R4,0‘;‘ ENDIF ‘;
                  ‘SUB R2,R4,R5‘;‘ IF ‘;‘ IF ‘;‘ADDI R2,R4,0‘;
                  ‘ ENDIF ‘;‘ ENDIF ‘;‘ADDI R3,R3,1‘]
                 ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
                  0 ≤ R3 ∧
                  ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
                  (¬(R3 = 0) ⟹  R4 IN_B rhoD))
```

In session box 23, `mk_while` is applied, returning the theorem displayed in Figure 7.29.

$\{Inv\}$
while $R_3 \neq 3$ **do**
receive R_2
if $R_3 \neq 0$ **then** `ADDI` R_2, R_4, 0 **endif**
`SUB` R_2, R_4, R_5
if $R_3 \neq 0$ **then**
if $R_4 < R_2$ **then**
`ADDI` R_2, R_4, 0
endif
endif
`ADDI` R_3, R_3, 1
$\{Inv \wedge \neg\neg(R_3 = 3)\}$

Figure 7.29: Proof Outline for HOL Session Box 23

Note that the conjunct "$\neg\neg(R_3 = 3)$" is attached to the postcondition of this theorem. Recall that the postcondition of our goal theorem (see Figure 7.9) is:

$$(\forall M\ (M \in \rho_D) \implies (M \leq R4)) \wedge \quad (R4 \in \rho_D)$$

Since $R_3 = 3$, $R_3 \neq 0$, so by the invariant Inv, $R_4 \in \rho_D$. Thus, the desired postcondition clearly follows from this result. All that is necessary to complete this proof is to apply `help_r_conseq`, demonstrate the appropriate implication, and apply `r_conseq`—seemingly a quite simple few steps. However, this simplicity is deceptive. Although *trivial*, the fact that $R_3 = 3$ implies $R_3 \neq 0$ is *not* trivial to prove in in HOL. In contrast to other theorem-proving systems (notably EHDM [11] and PVS [22]), the HOL system lacks automation—even trivialities such as $3 \neq 0$ must be demonstrated. $3 \neq 0$ is a theorem in the *decidable* theory of Presburger arithmetic. Theorem-proving systems frequently make use of *decision procedures* to relieve users of the burden of demonstrating the obvious, whereas HOL does not. $3 \neq 0$ must be demonstrated from Peano's axioms of arithmetic in HOL—a terribly laborious process. This proof has been left incomplete *intentionally* to emphasize the difficulty that this lack of automated support in HOL presents. To use this system to verify code, the user would frequently be sidetracked proving utterly trivial theorems in HOL such as $3 \neq 0$. Code verification is a difficult task even with automated support, and HOL's lack of automation hinders the task severely.

```
#let rec_proc_thm = mk_while loop_thm;;                                    23
rec_proc_thm =
........ ⊢ Prov
           ((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
             0 ≤ R3 ∧
             ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
             (¬(R3 = 0) ⟹  R4 IN_B rhoD))
            [‘ WHILE ‘;‘ receive R2 ‘;‘ IF ‘;‘ADDI R2,R4,0‘;
             ‘ ENDIF ‘;‘SUB R2,R4,R5‘;‘ IF ‘;‘ IF ‘;
             ‘ADDI R2,R4,0‘;‘ ENDIF ‘;‘ ENDIF ‘;‘ADDI R3,R3,1‘;
             ‘ ENDWHILE ‘]
             (((∀M. M IN_B rhoD ⟹  M ≤ R4) ∧
             0 ≤ R3 ∧
             ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
             (¬(R3 = 0) ⟹  R4 IN_B rhoD)) ∧
             ¬¬(R3 = 3))
```

Chapter 8

Conclusions

8.1 Summary

This thesis has presented an HOL mechanization of the axiomatic semantics of a simple distributed programming language. The language has features in support of sequential program execution in addition to asynchronous *send* and synchronous *receive* through which processes can communicate via message passing. We had hoped to mechanize a denotational semantics (extending Gordon's work), but were unable to find an approach for dealing with the absence of a global state in an asynchronous distributed program. Our semantics for the distributed features of the language derive from Schlichting's axiomatic semantics. Our mechanization package includes help functions which assist the user in applying inference rules to complicated Hoare triples. Among the benefits of our approach are:

- ease of modification—new rules can be readily added to the system.
- ease of use—help functions eliminate much user frustration and error, and the HOL tactics package can be applied directly to pre- and postconditions as needed. Also, large libraries of pre-proven theorems are available to the user in HOL.
- familiarity—someone relatively innocent of HOL can look at an HOL session transcript of a derivation and understand what has been demonstrated.

Our semantics for *send* and *receive* include *non-interference* assumptions, stating that the execution of these statements are atomic with respect to the objects they manipulate. The objects in question are machine registers and the send and receive bags, which record the messages sent to and received by a process, respectively. These assumptions would incur proof obligations on a verified implementation of *send* and *receive*. We are developing an implementation of these operations using a RISC-based microprocessor. At a higher level, we developing an operating system that is an extension of KIT [3]. Our operating system will allow processes (on a single host or on different hosts) to communicate only according to a security policy. Furthermore, the operating system will provide run-time support for a distributed programming language with higher-level constructs such as rendezvous and remote procedure call.

The target language will certainly be expanded as necessary to complete the application projects underway, and the tools to do so have been developed and tested in this thesis. The help functions (e.g., `mk_Sat_rec`) free the user from performing tedious and complicated syntactic operations like simultaneous substitution, thereby eliminating much human error and effort. Proofs in this system are created and are represented in a manner similar to proofs by hand, which renders them more accessible to the HOL novice.

8.2 Lessons Learned

Constructing large proofs in this system is difficult and time-consuming. I attribute this to several factors:

- the inherent difficulty of proving Hoare formulas,
- the requisite care necessary to use Schlichting's semantics,
- the severely limited automation in the HOL theorem proving system,
- and the need for an interface to the system to relieve the user of the repetitive aspects of proof organization.

Anyone who has performed proofs in Hoare logics knows that the process can be frustrating. Developing invariants and demonstrating correctness consume considerable effort on a time-unit per lines-of-code-verified basis. The semantics of **send** and **receive** can be particularly tricky to use. One especially valuable rule of thumb for developing proofs with Schlichting's rules is that global invariants for the communicating processes should be created which characterize the system as strongly and completely as possible.

The global invariant for the "simple" example of Chapter 7 required that *any* message sent to receiving process D had value 16 (see Figure 7.1). That is stronger than saying, for example, that "a message with value 16 was or will be sent to process D." One could not conclude from this that process D received 16 because it is possible with this invariant that some other process sent a message to D with value 8. It was necessary to strengthen this to the invariant which did work. Developing the system invariant for the second example of Chapter 7 proved to be a similar experience, although it was more difficult due to the increased complexity of the code.

Unfortunately, automating the process does not make it much easier. Generally speaking, humans are much better "provers" than are machines. Verification with an automated theorem prover offers greater assurance that the verification is correct—humans do not check proofs as well as a computer can. Applying the inference rules, axiom schemata and help functions is quite easy, and in many ways the user is relieved of the burden of performing multitudes of repetitive syntactic operations like simultaneous substitution. In particular, constructing the $Sat(r)$ sentence for a **receive** statement is an enormously tedious task which can be completely avoided through application of the help function `mk_Sat_rec`. However, the effort in demonstrating a Hoare formula increases considerably when performed with an automated theorem prover. Proving the "linking" theorems—those theorems purely of the assertion language occurring in the hypotheses of Hoare inference rules— involves the greatest increase in effort. Greater effort is required to prove these theorems because an automated theorem prover must be *guided in steps* through a proof by a human user.

The size of the "steps" which a theorem proving system can take vary from one system to another. The size of the proof steps achievable by the HOL system is quite small indeed.

"Why", the astute reader may well wonder, "should I have to prove that $3 \neq 0$ as in the last example of Chapter 7?" Certainly, the fact that $3 \neq 0$ has little (apparently) to do with code verification. With the HOL system, one is left to ponder this mysterious relationship between 3, 0, and code verification as one proves $3 \neq 0$ from Peano's axioms.

This dearth of automation in HOL seriously hindered the demonstration of the linking theorems in the examples of Chapter 7. All of the linking theorems in these examples were *trivial* facts about bags and arithmetic, yet large amounts of time had to be spent in their proving. The author is, admittedly, not an HOL guru, and some of this difficulty must surely be attributed to lack of experience with the HOL system. However, his literally years of experience constructing proofs in Mathematics and Logic (of considerably greater complexity than those found in Chapter 7) did little to expedite proofs in HOL. The human user of HOL typically understands why his proof goal is true, yet he must (not infrequently) construct parts of proofs at the "modus ponens" level; thus, using HOL practically *guarantees* that considerable effort must be spent on all but the most trivial theorems.

The pre- and postconditions of a Hoare formula will frequently involve arithmetic relationships, and arithmetic facts are notoriously tedious to demonstrate in HOL. Clearly, a suite of arithmetic decision procedures would greatly enhance the value of HOL as a verification tool. Other theorem proving systems (e.g., EHDM [11] and the Boyer-Moore prover [5]) have them, so the decision procedures themselves are already available.

8.3 Future Work

In order for this system to become useful to the "UCD Tower", its target language must be enriched to a language appropriate for writing an operating system kernel. So, more general assembly-level instructions should be added. The semantics of these instructions will have to be devised. Also, the semantics of the target language should be brought closer to that of a "real" assembly language. Recall that a number of simplifying assumptions were made about the instructions mechanized in this thesis which make the language unlike an assembly language (e.g., registers are integers, memory is ignored, etc). So in order to make this language more "real", more instructions should be added, and operand types

closer to actual registers should be defined.

This work for this thesis developed more than an integration into HOL of a particular axiomatic semantics. Rather, we created the necessary HOL machinery for specifying partial correctness conditions as Hoare triples in HOL. That is, axiom schemas such as those of $\mathcal{PL}$ may be readily mechanized in HOL with the ML and HOL constructs used here to mechanize $\mathcal{PL}$. State-dependent behavior is frequently specified with Hoare triples, and thus this package could prove quite useful. In particular, we believe the axiomatic semantics for "higher-level" programming language constructs such as the Ada *rendezvous*, the input and output commands of CSP, and the operations of SR may be mechanized readily.

Appendix A

hoare_defs.ml

```
system `/bin/rm hoare.th`;;

new_theory `hoare`;;

loadf `/usr/home/harrison/hol/Library/ind-defs/ind-defs`;;

load_library `ind_defs`;;

load_library `string`;;

load_library `bags`;;

%--------------- new definitions for bags ----------------------%

let SUB_B_DEF = new_infix_definition
       (`SUB_B_DEF`,
       "SUB_B (x:*)  (b:(*)bag) =
         ABS_bag (λy. ((y=x) →
               (((REP_bag b y) = 0) →  0 | (REP_bag b y) - 1) |
               REP_bag b y))");;

let DIFF_B_DEF = new_infix_definition
       (`DIFF_B_DEF`,
       "DIFF_B (b:(*)bag) (c:(*)bag) =
         ABS_bag (λx. ((REP_bag b x) ≥ (REP_bag c x)) →
                                          (REP_bag b x) - (REP_bag c x) |
     0)");;

%-------------- Rules of Inference ------------------%

let IS_AXIOM = new_definition( `IS_AXIOM`,
        "∀(phi1:bool) (sl:(string)list) (phi2:bool).
                (IS_AXIOM phi1 sl phi2) = T");;

let obvious_thm = prove_thm(`obvious_thm`,
 "∀phi:bool. (phi = T) ⟹  (phi)",
```

```
 ASM_REWRITE_TAC[] );;

let SIMPLE =
         let thm1 = SPEC "IS_AXIOM t1 st t2" obvious_thm in
         let thm2 = SPEC ("t2:bool")
             (SPEC ("st:(string)list") (SPEC ("t1:bool") IS_AXIOM)) in
         let thm3 = MP thm1 thm2 in
                    GEN "t1" (GEN "st" (GEN "t2" thm3));;


let (Prov_rules,Prov_ind) =
   let Prov = "Prov:bool->(string)list->bool->bool"
in
   new_inductive_definition false `Prov`
   ("^Prov t1 st t2", [])
%Axiom Introduction%
   [ [    "IS_AXIOM (t1:bool) (st:(string)list) (t2:bool):bool"] ,
         %-------------------------------------------------------%
          "^Prov t1 st t2" ;
%Composition Rule%
     [     "^Prov (t1:bool) (st1:(string)list) (t2:bool)";
           "^Prov (t3:bool) (st2:(string)list) (t4:bool)";
           "t2==> t3" ],
         %-------------------------------------------------------%
           "^Prov t1 (APPEND st1 st2) t4" ;
%Left Consequence%
     [     "^Prov (pre:bool) (st:(string)list) (post:bool)";
           "phi ==>  pre"],
         %-------------------------------------------------------%
           "^Prov phi st post";
%Right Consequence%
     [     "^Prov (pre:bool) (st:(string)list) (post:bool)";
           "post ==>  phi"],
         %-------------------------------------------------------%
           "^Prov pre st phi";
%IF rule for forward jumps%
     [     "^Prov (P/\ B) (st:(string)list) (Q:bool)";
           "P/\ ~B==> Q" ],
```

```
        %-------------------------------------------------------------%
          "^Prov P (APPEND [‘ IF ‘] (APPEND st [‘ ENDIF ‘] )) Q" ;
%WHILE rule for backwards jumps%
     [    "^Prov (Inv∧ B) (st:(string)list) (Inv:bool)" ],
        %-------------------------------------------------------------%
          "^Prov (Inv)
               (APPEND [‘ WHILE ‘] (APPEND st [‘ ENDWHILE ‘])) (Inv∧ ¬B)"  ];;

let Axiom_intro = (el 1 Prov_rules);;

let Composition_rule = (el 2 Prov_rules);;

let Left_consequence_rule = (el 3 Prov_rules);;

let Rt_consequence_rule = (el 4 Prov_rules);;

let IF_rule = (el 5 Prov_rules);;

let WHILE_rule = (el 6 Prov_rules);;

let sim_sub =
(λwff:term. λtarget:term. λexpr:term.
         snd( dest_thm (INST [(expr,target)] (mk_thm([],wff)))));;

let compose th1 th2 th3 =
let b1 = snd(dest_comb(fst(dest_comb(fst (dest_comb(snd(dest_thm(th1)))))))) in
let s1 = snd(dest_comb(fst (dest_comb(snd(dest_thm(th1)))))) in
let b2 = snd (dest_comb(snd(dest_thm(th1)))) in
let b3 = snd(dest_comb(fst(dest_comb(fst (dest_comb(snd(dest_thm(th2)))))))) in
let s2 = snd(dest_comb(fst (dest_comb(snd(dest_thm(th2)))))) in
let b4 = snd (dest_comb(snd(dest_thm(th2)))) in
let t1 = snd (dest_thm(th1)) in
let t2 = snd (dest_thm(th2)) in
let t3 = snd (dest_thm(th3)) in
let all_conj = mk_conj(t1,mk_conj(t2,t3)) in
let lm1 = DISCH t1 (DISCH t2 (DISCH t3 (ASSUME all_conj))) in
let lm2 = MP(MP(MP lm1 th1) th2) th3 in
let lm3 = SPEC b4 (SPEC s2 (SPEC s1 (SPEC b1 Composition_rule))) in
```

```
let hyp1 = snd(dest_exists(fst(dest_imp(snd(dest_thm(lm3)))))) in
let form1 =sim_sub hyp1 "t2" b2 in
let lm4 = EXISTS (form1,b3) lm2 in
let hyp2 = fst(dest_imp(snd(dest_thm(lm3)))) in
let lm5 = EXISTS (hyp2,b2) lm4 in
        REWRITE_RULE [APPEND] (MP lm3 lm5);;

let help_compose th1 th2 =
     let post1 = snd(dest_comb(snd(dest_thm(th1)))) in
     let pre2  =
       snd(dest_comb(fst(dest_comb(fst(dest_comb(snd(dest_thm(th2)))))))) in
     let help_thm = mk_imp(post1,pre2) in
           set_goal([], help_thm);;

let l_conseq th1 th2 =
let b1 = snd(dest_comb(fst(dest_comb(fst (dest_comb(snd(dest_thm(th1)))))))) in
let s1 = snd(dest_comb(fst (dest_comb(snd(dest_thm(th1)))))) in
let b2 = snd (dest_comb(snd(dest_thm(th1)))) in
let b3 = fst (dest_imp(snd(dest_thm(th2)))) in
let t1 = snd (dest_thm(th1)) in
let t2 = snd (dest_thm(th2)) in
let all_conj = mk_conj(t1,t2) in
let lm1 = DISCH t1 (DISCH t2 (ASSUME all_conj)) in
let lm2 = MP(MP lm1 th1) th2 in
let lm3 = SPEC b3 (SPEC b2 (SPEC s1 Left_consequence_rule)) in
let hyp = fst(dest_imp(snd(dest_thm(lm3)))) in
       MP lm3 (EXISTS (hyp,b1) lm2);;

let help_l_conseq th1 new_pre =
     let prec  =
       snd(dest_comb(fst(dest_comb(fst(dest_comb(snd(dest_thm(th1)))))))) in
     let help_thm = mk_imp(new_pre,prec) in
           set_goal([], help_thm);;

let r_conseq th1 th2 =
let b1 = snd(dest_comb(fst(dest_comb(fst (dest_comb(snd(dest_thm(th1)))))))) in
let s1 = snd(dest_comb(fst (dest_comb(snd(dest_thm(th1)))))) in
let b2 = snd (dest_comb(snd(dest_thm(th1)))) in
```

```
let b3 = fst (dest_imp(snd(dest_thm(th2)))) in
let b4 = snd (dest_imp(snd(dest_thm(th2)))) in
let t1 = snd (dest_thm(th1)) in
let t2 = snd (dest_thm(th2)) in
let all_conj = mk_conj(t1,t2) in
let lm1 = DISCH t1 (DISCH t2 (ASSUME all_conj)) in
let lm2 = MP(MP lm1 th1) th2 in
let lm3 = SPEC b4 (SPEC s1 (SPEC b1 Rt_consequence_rule)) in
let hyp = fst(dest_imp(snd(dest_thm(lm3)))) in
       MP lm3 (EXISTS (hyp,b2) lm2);;

let help_r_conseq th1 new_post =
     let post1 = snd(dest_comb(snd(dest_thm(th1)))) in
     let help_thm = mk_imp(post1,new_post) in
            set_goal([], help_thm);;

let mk_while th =
    let tmp = dest_comb(snd(dest_thm(th))) in
    let inv = snd(tmp) in
    let stmtlst = snd(dest_comb(fst(tmp))) in
    let b = snd(dest_conj(snd(dest_comb(fst(dest_comb(fst(tmp)))))))) in
    let impl = SPEC stmtlst (SPEC b (SPEC inv WHILE_rule)) in
        PURE_REWRITE_RULE [APPEND] (MP impl th);;

let mk_if th1 th2 =
    let tmp = dest_comb(snd(dest_thm(th1))) in
    let Q = snd(tmp) in
    let st = snd(dest_comb(fst(tmp))) in
    let P = fst(dest_conj(snd(dest_comb(fst(dest_comb(fst(tmp)))))))) in
    let B = snd(dest_conj(snd(dest_comb(fst(dest_comb(fst(tmp)))))))) in
    let trm1 = snd(dest_thm(thm1)) in
    let trm2 = snd(dest_thm(thm2)) in
    let help_conj = mk_conj (trm1,trm2) in
    let help_thm =
MP (MP (DISCH trm1 (DISCH trm2 (ASSUME help_conj))) th1) th2 in
    let spec_IF_rule = SPEC Q (SPEC st (SPEC P IF_rule)) in
    let hyp = fst(dest_imp(snd(dest_thm(spec_IF_rule)))) in
    let antec = EXISTS (hyp,B) help_thm in
```

```
        PURE_REWRITE_RULE [APPEND] (MP spec_IF_rule antec);;

% Warning: this implementation can not distinguish between upper and lower
  case letters. So for example, be careful not to have a program variable
  "m" and a quantified logical variable "M". The consequences can be
  annoying. %
let once_ssub wff target expr =
let rewrite_eqn = mk_eq(target,expr) in
let rewrite_thm = mk_thm([],rewrite_eqn) in
        let rewrite_wff = mk_thm([],wff) in
    snd(dest_thm(PURE_ONCE_REWRITE_RULE [rewrite_thm] rewrite_wff));;

let mk_assign operation post target expr =
        let pre = once_ssub post target expr in
        let th1 = SPEC post (SPEC operation (SPEC pre SIMPLE)) in
        let th2 = SPEC post (SPEC operation (SPEC pre Axiom_intro)) in
                MP th2 th1;;

new_constant(`NZ`,":bool");;

new_constant(`GE`,":bool");;

let mk_ADDI operation post target expr k =
        let new_expr = mk_comb(mk_comb("+",expr),k) in
        let pre = once_ssub post target new_expr in
        let conj1 = mk_eq("NZ",mk_neg(mk_eq(new_expr,"0"))) in
        let conj2 = mk_eq("GE",mk_comb(mk_comb("≥",new_expr),"0")) in
let new_post = mk_conj(post,mk_conj(conj1,conj2)) in
        let th1 = SPEC new_post (SPEC operation (SPEC pre SIMPLE)) in
        let th2 = SPEC new_post (SPEC operation (SPEC pre Axiom_intro)) in
                MP th2 th1;;

% performs target := expr1 - expr2 %
let mk_SUB operation post expr1 expr2 target =
        let new_expr = mk_comb(mk_comb("-",expr2),expr1) in
        let pre = once_ssub post target new_expr in
        let conj1 = mk_eq("NZ",mk_neg(mk_eq(expr2,expr1))) in
        let conj2 = mk_eq("GE",mk_comb(mk_comb("≥",expr2),expr1)) in
```

```
let new_post = mk_conj(post,mk_conj(conj1,conj2)) in
        let th1 = SPEC new_post (SPEC operation (SPEC pre SIMPLE)) in
        let th2 = SPEC new_post (SPEC operation (SPEC pre Axiom_intro)) in
                 MP th2 th1;;

let mk_send W expr sigD =
        let sigD_PLUS_expr = mk_comb(mk_comb(mk_const
                 (`INSERT_B`,":num->((num)bag->(num)bag)"),expr), sigD) in
let Wprime = once_ssub W sigD sigD_PLUS_expr in
        let operation = "[` send `]" in
        let th1 = SPEC W (SPEC operation (SPEC Wprime SIMPLE)) in
        let th2 = SPEC W (SPEC operation (SPEC Wprime Axiom_intro)) in
                 MP th2 th1;;

let mk_Sat_rec R Q m MTEXT sD rD =
        let sD_diff_rD = mk_comb(mk_comb(mk_const
                 (`DIFF_B`,":(num)bag->((num)bag->(num)bag)"),sD), rD) in
        let conj2 = mk_comb(mk_comb(mk_const
                   (`IN_B`,":num->((num)bag->bool)"),MTEXT),sD_diff_rD) in
        let hypoth = mk_conj(R,conj2) in
        let rD_PLUS_MTEXT = mk_comb(mk_comb(mk_const
                 (`INSERT_B`,":num->((num)bag->(num)bag)"),MTEXT), rD) in
        let Qtmp = once_ssub Q m MTEXT in
        let Qprime = once_ssub Qtmp rD rD_PLUS_MTEXT in
let Sat_R = mk_imp(hypoth,Qprime) in
    set_goal([],Sat_R);;

let mk_receive t m Q =
let Qprime = snd(dest_imp(snd(dest_thm(t)))) in
let R = fst(dest_conj(fst(dest_imp(snd(dest_thm(t)))))) in
let conjunct2 = snd(dest_conj(fst(dest_imp(snd(dest_thm(t)))))) in
let label1 =
    fst(dest_const(fst(dest_comb(fst(dest_comb(conjunct2)))))) in
        let MTEXT = if (label1 = `IN_B`) then
                      snd(dest_comb(fst(dest_comb(conjunct2))))
                    else
       failwith `SAT(r) stated incorrectly` in
let label2 = fst(dest_const(fst(dest_comb(fst(
```

```
                dest_comb(snd(dest_comb(conjunct2)))))))) in
        let sigD =  if (label2 = `DIFF_B`) then
                  snd(dest_comb(fst(dest_comb(snd(dest_comb(conjunct2))))))
                else
       failwith `SAT(r) incorrectly stated` in
        let rhoD = snd(dest_comb(snd(dest_comb(conjunct2)))) in
let Qtmp = once_ssub Q m MTEXT in
% Constructs "MTEXT INSERT_B rhoD" %
        let rhoD_PLUS_MTEXT = mk_comb(mk_comb(mk_const
                 (`INSERT_B`,":num->((num)bag->(num)bag)"),MTEXT), rhoD) in
        let Qnew = once_ssub Qtmp rhoD rhoD_PLUS_MTEXT in
% The following contains a kluge to check whether the postcondition Q
  given by the user is justified by the satisfaction theorem given.
  Qnew - obtained from Q by simultaneously substituting MTEXT and
  (rhoD + MTEXT) for m and rhoD, respectively - should be identical to
  Qprime, which is the consequent of the given satisfaction theorem.
  MP (modus ponens) will evaluate successfully when applied to klugeTHM
  and Qnew if and only if Qnew and Qprime are _identical_. I used this
  trick because it is a painless method of determining whether Qnew and
  Qprime are *syntactically* identical. HOL already has this facility,
  although it appears to be not directly available to the user. Hence,
  the kluge was necessary.  %
        let klugeTHM = DISCH Qprime (ASSUME Qprime) in
% klugeEVAL evaluates successfully iff Qnew is identical to Qprime %
        let klugeEVAL = MP klugeTHM (mk_thm([],Qnew)) in
        let operation = "[` receive `]" in
        let th1 = SPEC Q (SPEC operation (SPEC R SIMPLE)) in
        let th2 = SPEC Q (SPEC operation (SPEC R Axiom_intro)) in
                 MP th2 th1;;
```

Appendix B

example1

```
%-------------------- example proof simple code----------------------%

%
            proof of the following outline

 ((sigD = EMPTY_BAG) ∧ Inv )
         A::       R_A := 16;
                   send R_A to D
 (16 IN_B sigD ∧
       (sigD = 16 INSERT_B EMPTY_BAG) ∧ Inv ∧
       (R_A = 16))

 ((rhoD = EMPTY_BAG) ∧ Inv ))
         D::       receive R_D
 (R_D = 16) ∧ Inv
                     where Inv is (∀M. M IN_B sigd ⟹  (M = 16)).

%

let sigD_DEF = new_definition(‘sigD_DEF‘,
               "sigD = (ABS_bag λx:num. 0)");;

let rhoD_DEF = new_definition(‘rhoD_DEF‘,
               "rhoD = (ABS_bag λx:num. 0)");;

new_constant(‘R_A‘,":num");;

new_constant(‘R_A_16‘,":num");;

new_constant(‘R_D‘,":num");;

new_constant(‘MTEXT‘,":num");;

let th0 =
let th = mk_ADDI "[‘ADDI R_A_16,R_A,0‘]"
```

```
   "(sigD = EMPTY_BAG)∧ (∀M:num. (M IN_B sigD) ⟹
    (M = 16))∧ ((R_A=16)∧ (R_A_16=16))"
          "R_A" "R_A_16" "0" in
                          REWRITE_RULE [] th;;

help_r_conseq th0
        "(sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧
         (R_A = 16)";;

e(DISCH_TAC THEN ASM_REWRITE_TAC []);;

let th0 = r_conseq th0 (top_thm());;

let th1 = mk_send "[‘ send R_A to D ‘]"
"(16 IN_B sigD) ∧
(sigD = (16 INSERT_B EMPTY_BAG)) ∧ (∀M:num. (M IN_B sigD) ⟹  (M = 16)) ∧
(R_A = 16)"
                               "R_A" "sigD";;

set_goal([],
"((sigD = EMPTY_BAG) ∧ (R_A = 16) ∧ (∀M. (M IN_B sigD) ⟹  (M=16))) ⟹
             ((16 IN_B ( R_A INSERT_B sigD)) ∧
              ((R_A INSERT_B sigD) = (16 INSERT_B EMPTY_BAG)) ∧
              (∀M. (M IN_B (R_A INSERT_B sigD)) ⟹  (M = 16)) ∧
              (R_A = 16))");;

e(DISCH_TAC THEN
CONJ_TAC THEN
REWRITE_TAC [IN_B]);;

e(ASM_REWRITE_TAC [(DISCH "R_A=16" (ASSUME "16 = R_A"))] );;

e(CONJ_TAC THEN
ASM_REWRITE_TAC [(DISCH "R_A=16" (ASSUME "16 = R_A"))] THEN
ASM_REWRITE_TAC [] THEN
REWRITE_TAC [IN_B]);;

let help_thm0 = top_thm();;
```

```
let th2 = l_conseq th1 help_thm0;;

let comp_thm0 = DISCH
"((sigD = EMPTY_BAG) ∧ (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A = 16))"
(ASSUME "(sigD = EMPTY_BAG) ∧ (R_A=16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))");;

help_compose th0 th2;;

e(DISCH_TAC THEN ASM_REWRITE_TAC []);;

let comp_thm0 = top_thm();;

let th3 = compose th0 th2 comp_thm0;;

help_l_conseq th3 "((sigD = EMPTY_BAG) ∧
                   (∀M. M IN_B sigD ⟹  (M = 16)) ∧ (R_A_16 = 16))";;

e(DISCH_TAC THEN ASM_REWRITE_TAC [(SPEC "16" ADD_0)]);;

let th4 = l_conseq th3 (top_thm());;

% DIFF_B_LEMMA %
set_goal([],"∀x b c. (x IN_B (b DIFF_B c)) ⟹  (x IN_B b)");;

e(STRIP_TAC THEN STRIP_TAC THEN STRIP_TAC);;

e(REWRITE_TAC [DIFF_B_DEF; IN_B_DEF; R_A]);;

e(REWRITE_TAC[(BETA_CONV "(λx.
     ((REP_bag b x) ≥ (REP_bag c x) →
      (REP_bag b x) − (REP_bag c x) |
      0)) x")]);;

e(ASM_CASES_TAC "(REP_bag b x = 0)");;

e(REWRITE_TAC[GREATER_OR_EQ]);;
```

```
e(ASM_REWRITE_TAC[]);;

e(ASM_CASES_TAC "0 = (REP_bag c x)");;

e(ASM_REWRITE_TAC[SUB_EQUAL_0]);;

e(ASM_REWRITE_TAC[]);;

e(ASSUM_LIST(λthl. ASSUME_TAC
           (REWRITE_RULE [LESS_OR_EQ; (el 1 thl)]
          (SPEC "REP_bag c x" ZERO_LESS_EQ))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
     (REWRITE_RULE [(el 1 thl)]
                      (SPEC "REP_bag c x" (SPEC "0" LESS_ANTISYM)))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
     (REWRITE_RULE [(SPEC "0" (SPEC "REP_bag c x" GREATER))]
           (el 1 thl))));;

e(ASM_REWRITE_TAC [GREATER]);;

e(ASM_REWRITE_TAC []);;

let DIFF_B_LEMMA = save_thm(`DIFF_B_LEMMA`,(top_thm()));;

backup();;

e(ASSUM_LIST(λthl. ASSUME_TAC
             (ONCE_REWRITE_RULE [EQ_SYM_EQ] (el 1 thl))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
             (REWRITE_RULE [(el 1 thl)] (SPEC "(REP_bag c x)" num_CASES))));;




%---------------- mk_SAT_rec example (I believe) --------%
```

```
let rhoD_DEF = new_definition(‘rhoD_DEF‘,
               "rhoD = (ABS_bag λx:num. 0)");;

mk_Sat_rec
"(rhoD = EMPTY_BAG) ∧ (∀M:num. (M IN_B sigD) ⟹  (M = 16))"
"(R_D=16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))"
"R_D" "MTEXT" "sigD:(num)bag" "rhoD:(num)bag";;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

e(ASSUM_LIST (λthl. ASSUME_TAC
              (REWRITE_RULE [(el 1 thl)]
       (SPEC "rhoD" (SPEC "sigD" (SPEC "MTEXT"
 (INST_TYPE [":num",":*"] DIFF_B_LEMMA)))))));;

e(ASSUM_LIST(λthl. ASM_REWRITE_TAC
     [(MP (SPEC "MTEXT" (el 3 thl)) (el 1 thl))]));;

let sat_r = top_thm();;

let th0_D = mk_receive "[‘ receive R_D ‘]" sat_r "R_D"
         "(R_D=16) ∧ (∀M. M IN_B sigD ⟹  (M = 16))";;
```

Appendix C

example2

```
%---------- proof for second harder example ----%

new_constant(`R0`,":num");;

new_constant(`R1`,":num");;

new_constant(`R2`,":num");;

new_constant(`R3`,":num");;

new_constant(`R4`,":num");;

new_constant(`R5`,":num");;

new_constant(`mess`,":num");;

new_constant(`mess_A`,":num");;

new_constant(`mess_B`,":num");;

new_constant(`mess_C`,":num");;

let sigD_DEF = new_definition(`sigD_DEF`,
               "sigD = (ABS_bag λx:num. 0)");;

let rhoD_DEF = new_definition(`rhoD_DEF`,
               "rhoD = (ABS_bag λx:num. 0)");;

%---  LEMMAS  ---%

set_goal([], "∀b x y. (x IN_B (b SUB_B y)) ⟹  (x IN_B b)");;

e(REWRITE_TAC[IN_B_DEF;SUB_B_DEF;R_A]);;

e(STRIP_TAC THEN STRIP_TAC THEN STRIP_TAC);;
```

```
e(REWRITE_TAC [(BETA_CONV
"((λy'.
     ((y' = y) →
      ((REP_bag b y' = 0) →  0 | (REP_bag b y') − 1) |
      REP_bag b y'))
   x)")]);;

e(ASM_CASES_TAC "x = y");;

e(ASM_REWRITE_TAC []);;

e(ASM_CASES_TAC "REP_bag b y = 0");;

e(ASM_REWRITE_TAC []);;

e(ASM_REWRITE_TAC []);;

e(ASM_REWRITE_TAC []);;

let SUB_IN_B_LEMMA = save_thm(`SUB_IN_B_LEMMA`,(top_thm()));;

let lemma1 =
prove_thm ( `lemma1` ,"∀n. (¬(n=0) ∧ (0 ≤ n)) ⟹  (0 ≤ (n−1))",
   INDUCT_TAC THEN
   REWRITE_TAC[] THEN
   STRIP_TAC THEN
   REWRITE_TAC[SUC_SUB1] THEN
   REWRITE_TAC [LESS_OR_EQ] THEN
   DISJ_CASES_TAC (SPEC "n:num" LESS_0_CASES) THEN
   ASM_REWRITE_TAC [] THEN
   ASM_REWRITE_TAC []);;

let lemma3 = prove_thm(`lemma3`,"∀t1 t2. t1 ⟹  (t2 ⟹  (t1 ∧ t2))",
       STRIP_TAC THEN
       STRIP_TAC THEN
       DISCH_TAC THEN
       DISCH_TAC THEN
```

```
       CONJ_TAC THEN
       ASM_REWRITE_TAC [] THEN
       ASM_REWRITE_TAC []);;

let lemma4 = prove_thm(`lemma4`,"∀t1 t2. t1 ⟹ (t2 ⟹ (t1 ⟹ t2))",
                            REPEAT STRIP_TAC THEN ASM_REWRITE_TAC []);;

let lemma5 =
     prove_thm(`lemma5`,"∀m n. m < n ⟹ m ≤ n",
           REPEAT STRIP_TAC THEN ASM_REWRITE_TAC [LESS_OR_EQ]);;

let lemma6 = prove_thm(`lemma6`,
"∀rho reg .
((∃M1. M1 IN_B rho ∧ (∀M2. M2 IN_B rho ⟹ M2 ≤ M1) ∧ (M1 = reg)) ⟹
 (∃M1. (M1 IN_B rho) ∧ ((∀M2. M2 IN_B rho ⟹ M2 ≤ M1))))",
                  REPEAT STRIP_TAC THEN
                  EXISTS_TAC "M1" THEN
                  ASM_REWRITE_TAC []);;

let lemma7 =
prove_thm(`lemma7`,"(∀b r. (∀ M2. (M2 IN_B (b SUB_B r)) ⟹ (M2 ≤ r)) ⟹
                     (∀ M2. (M2 IN_B b) ⟹ (M2 ≤ r)))",
           REPEAT STRIP_TAC THEN
           ASM_CASES_TAC "r = M2" THENL
           [ %1st subgoal%
                REWRITE_TAC [LESS_OR_EQ] THEN
                ASM_REWRITE_TAC [];
             %2nd subgoal%
                ASSUM_LIST(λthl. ASSUME_TAC
                (SPEC "r" (SPEC "M2" (SPEC "b"
                      (INST_TYPE [":num",":*"] IN_SUB_B_THM))))) THEN
                ASSUM_LIST(λthl. ASSUME_TAC
                  (REWRITE_RULE [(el 2 thl);(el 3 thl)] (el 1 thl))) THEN
                ASSUM_LIST(λthl. ASSUME_TAC
                  (REWRITE_RULE [(el 1 thl)]
                      (SPEC "M2" (el 5 thl)))) THEN
                ASM_REWRITE_TAC []]);;
```

```
let lemma8 =
prove_thm(`lemma8`,"∀t1 t2 t3. t1 ⟹ (t2 ⟹ (t3 ⟹ (t1 ∧ t2 ∧ t3)))",
        REPEAT STRIP_TAC THEN ASM_REWRITE_TAC []);;

let lemma9 =
prove_thm(`lemma9`,"∀m n. (m = n) ⟹ ¬(n < m)",
    STRIP_TAC THEN STRIP_TAC THEN STRIP_TAC THEN
    ONCE_ASM_REWRITE_TAC [NOT_LESS] THEN
    ONCE_ASM_REWRITE_TAC [] THEN
    ONCE_ASM_REWRITE_TAC [LESS_EQ_REFL]);;

let lemma10 = prove_thm(`lemma10`,"∀t. F ⟹ t", REPEAT STRIP_TAC);;


let Inv = "(∀M. (M IN_B rhoD) ⟹ (M ≤ R4)) ∧
           (0 ≤ R3) ∧
           ((R3 = 0) ⟹ (rhoD = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹ (R4 IN_B rhoD))";;

let Q = "(∀M. (M IN_B (rhoD SUB_B R2)) ⟹ (M ≤ R4)) ∧
           (0 ≤ R3) ∧ (R2 IN_B rhoD) ∧
           ((R3 = 0) ⟹ ((rhoD SUB_B R2) = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹ (R4 IN_B (rhoD SUB_B R2)))";;

let Inv' = mk_conj(Inv,"¬(R3 = 3)");;

let Q' = mk_conj(Q,"¬(R3 = 3)");;

%----- proof of th1 -----%

mk_Sat_rec
Inv'
Q'
"R2" "mess" "sigD:(num)bag" "rhoD:(num)bag";;

e(STRIP_TAC);;

e(ASM_REWRITE_TAC [(SPEC "mess" (SPEC "rhoD"
```

```
            (INST_TYPE [":num",":*"] INSERT_SUB_B_LEMMA)))]);;

e(REWRITE_TAC [COMPONENT_B]);;


%- end proof of th1 -----%

let sat_thm = top_thm();;

let th1 = mk_receive "[` receive R2 `]" sat_thm "R2" Q';;

% proof of th2 %

let Q_if1 = mk_conj(Q',"R3 = 0");;

let new_post = "(∀M. (M IN_B (rhoD SUB_B R2)) ⟹  (M ≤ R4)) ∧
        (0 ≤ R3) ∧ (R2 IN_B rhoD) ∧
        ((R3 = 0) ⟹
        ((R2 = R4) ∧ (R4 IN_B rhoD) ∧ ((rhoD SUB_B R2) = EMPTY_BAG))) ∧
        (¬(R3 = 0) ⟹  (R4 IN_B (rhoD SUB_B R2))) ∧ ¬(R3 = 3)";;

let th2 = mk_ADDI "[`ADDI R2,R4,0`]" new_post "R4" "R2" "0";;

help_l_conseq th2 Q_if1;;

e(STRIP_TAC THEN ASM_REWRITE_TAC [ADD_0]);;

e(ASSUME_TAC (SPEC "(rhoD SUB_B R2)" (SPEC "M"
                (INST_TYPE [":num",":*"] (MEMBER_IMP_NONEMPTY_BAG)))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
            (MP (el 5 thl) (el 2 thl))));;

e(ASM_REWRITE_TAC [MEMBER_IMP_NONEMPTY_BAG;IN_B]);;

let lc_thm_th2 = top_thm();;

let th2 = l_conseq th2 lc_thm_th2;;
```

```
let Q_ifnot1 = mk_conj(Q',"¬(R3 = 0)");;

help_r_conseq th2 new_post;;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let th2 = r_conseq th2 (top_thm());;

set_goal([],"^Q_ifnot1 ⟹  ^new_post");;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let if_thm1 = top_thm();;

let th2 = mk_if th2 (top_thm());;

% th3 %

help_compose th1 th2;;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let th3 = compose th1 th2 (top_thm());;

% th4 %

let th4 = mk_SUB "[`SUB R2,R4,R5`]" new_post "R2" "R4" "R5";;

help_compose th3 th4;;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let th4 = compose th3 th4 (top_thm());;

th4;;

% th5 %
```

```
let Inv_spec = "(∀M. (M IN_B rhoD) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧
           ((R3 = 0) ⟹  (rhoD = EMPTY_BAG)) ∧
           (¬(R3 = 0) ⟹  (R4 IN_B rhoD)) ∧
           (R4 IN_B rhoD) ∧ ¬(R3 = 3)";;

let Q_outerif = mk_conj(new_post,"¬(R3=0)");;

let Q_innerif = mk_conj(Q_outerif,"R4 < R2");;

let th5 = mk_ADDI "[`ADDI R2,R4,0`]" Inv_spec "R4" "R2" "0";;

help_l_conseq th5 Q_innerif;;

e(STRIP_TAC THEN ASM_REWRITE_TAC [ADD_0]);;

% superfluous %
e(ASSUME_TAC (SPEC "R4" (SPEC "R2" lemma9)));;

% superfluous %
e(ASSUM_LIST(λthl. ASSUME_TAC
                       (REWRITE_RULE [(el 3 thl)] (el 1 thl))));;

% superfluous %
e(ASSUM_LIST(λthl. ASSUME_TAC
                       (REWRITE_RULE [(el 1 thl)] (el 7 thl))));;

% superfluous %
e(ASM_REWRITE_TAC []);;

e(STRIP_TAC THEN STRIP_TAC);;

e(ASM_CASES_TAC "R2 = M");;

e(ASM_REWRITE_TAC [LESS_OR_EQ]);;

e(ASSUM_LIST(λthl.  ASSUME_TAC
```

```
    (REWRITE_RULE [(el 2 thl);(el 1 thl)]
      (SPEC "R2" (SPEC "M" (SPEC "rhoD"
          (INST_TYPE [":num",":*"] IN_SUB_B_THM)))))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
            (SPEC "M" (el 11 thl))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
        (MP (el 1 thl) (el 2 thl))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
        (MP (SPEC "R2" (SPEC "R4" lemma5)) (el 6 thl))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
         (REWRITE_RULE [(el 1 thl);(el 2 thl)]
            (SPEC "R2" (SPEC "R4" (SPEC "M" LESS_EQ_TRANS))))));;

e(ASM_REWRITE_TAC []);;

let th5 = l_conseq th5 (top_thm());;

help_r_conseq th5 Inv_spec;;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let th5 = r_conseq th5 (top_thm());;

let Q_innernot = mk_conj(Q_outerif,"¬(R4 < R2)");;

set_goal([], "^Q_innernot ⟹  ^Inv_spec");;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

e(ASSUM_LIST(λthl. REWRITE_TAC
        [(MP
                  (SPEC "R2" (SPEC "R4" (SPEC "rhoD"
  (INST_TYPE [":num",":*"] SUB_IN_B_LEMMA))))
                  (MP (el 4 thl) (el 2 thl)))]));;
```

```
e(ASM_CASES_TAC "R4 = R2");;

e(ASM_REWRITE_TAC []);;

e(REPEAT STRIP_TAC);;

e(ASM_CASES_TAC "R2 = M");;

e(ASM_REWRITE_TAC [LESS_OR_EQ]);;

e(ASSUM_LIST(λthl.  ASSUME_TAC
    (REWRITE_RULE [(el 2 thl);(el 1 thl)]
      (SPEC "R2" (SPEC "M" (SPEC "rhoD"
          (INST_TYPE [":num",":*"] IN_SUB_B_THM)))))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
            (SPEC "M" (el 12 thl))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
        (MP (el 1 thl) (el 2 thl))));;

% necessary∃ %
e(ASSUM_LIST(λthl. ASSUME_TAC
        (MP (SPEC "R2" (SPEC "R4" lemma5)) (el 6 thl))));;

e(ASSUM_LIST(λthl. ASM_REWRITE_TAC
                 [(ONCE_REWRITE_RULE [(el 6 thl)] (el 1 thl))]));;

e(ASSUM_LIST(λthl. ASSUME_TAC
              (MP (el 5 thl) (el 3 thl))));;

e(REPEAT STRIP_TAC);;

e(ASM_CASES_TAC "R2 = M");;

e(ASSUM_LIST(λthl. ASSUME_TAC
         (REWRITE_RULE [(el 5 thl);(el 4 thl)]
```

```
            (SPEC "R2" (SPEC "R4" LESS_CASES_IMP)))));;

e(REWRITE_TAC [LESS_OR_EQ]);;

e(ASSUM_LIST(λthl. ASSUME_TAC
     (ONCE_REWRITE_RULE [EQ_SYM_EQ] (el 2 thl))));;

e(ASSUM_LIST(λthl. REWRITE_TAC
     [(PURE_ONCE_REWRITE_RULE [(el 2 thl)] (el 1 thl))]));;

e(ASSUM_LIST(λthl. PURE_ONCE_REWRITE_TAC
     [(el 2 thl)]));;

e(REWRITE_TAC []);;

e(ASSUM_LIST(λthl.  ASSUME_TAC
    (REWRITE_RULE [(el 2 thl);(el 1 thl)]
      (SPEC "R2" (SPEC "M" (SPEC "rhoD"
          (INST_TYPE [":num",":*"] IN_SUB_B_THM)))))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
            (SPEC "M" (el 13 thl)))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
        (MP (el 1 thl) (el 2 thl)))));;

e(ASM_REWRITE_TAC []);;

let inner_not_thm = top_thm();;

let th5 = mk_if th5 (top_thm());;

let safe_5 = th5;;

let th5 = safe_5;;

let Inv'' = "(∀M. (M IN_B rhoD) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧
```

```
          (¬(R3 = 0) ⟹  (R4 IN_B rhoD)) ∧
          (R4 IN_B rhoD) ∧
          ¬(R3 = 3)";;

help_r_conseq th5 Inv'';;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let th5 = r_conseq th5 (top_thm());;

help_l_conseq th5 Q_outerif;;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let th5 = l_conseq th5 (top_thm());;

let Q_outernot = mk_conj(new_post,"¬¬(R3 = 0)");;

set_goal([],"^Q_outernot ⟹  ^Inv''");;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

e(ASSUM_LIST(λthl. ASSUME_TAC
           (MP (el 4 thl) (REWRITE_RULE [] ((el 1 thl))))));;

e(ASM_REWRITE_TAC []);;

e(REPEAT STRIP_TAC);;

e(ASM_CASES_TAC "R2 = M");;

e(ASSUME_TAC (SPEC "M" (SPEC "R2"
         (INST_TYPE [":num",":*"] EQ_SYM))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
           (MP (el 1 thl) (el 2 thl)))));;

e(REWRITE_TAC [LESS_OR_EQ]);;
```

```
e(ASSUM_LIST(λthl. PURE_ONCE_REWRITE_TAC [(el 1 thl)]));;

e(ASSUM_LIST(λthl. PURE_ONCE_REWRITE_TAC [(el 5 thl)]));;

e(REWRITE_TAC []);;

e(ASSUM_LIST(λthl.  ASSUME_TAC
    (REWRITE_RULE [(el 2 thl);(el 1 thl)]
      (SPEC "R2" (SPEC "M" (SPEC "rhoD"
          (INST_TYPE [":num",":*"] IN_SUB_B_THM)))))));;

e(ASSUM_LIST(λthl. ASSUME_TAC
     (REWRITE_RULE [(el 4 thl);IN_B] (el 1 thl))));;

e(ASSUM_LIST(λthl. ASM_REWRITE_TAC
          [(MP (SPEC "M ≤ R4" lemma10) (el 1 thl))]));;

let outer_not_if_thm = top_thm();;

let th5 = mk_if th5 (top_thm());;

let Inv''' = "(∀M. (M IN_B rhoD) ⟹  (M ≤ R4)) ∧
           (0 ≤ R3) ∧
           (¬(R3 = 0) ⟹  (R4 IN_B rhoD)) ∧
           ¬(R3 = 0)";;

let th6 = mk_ADDI "[`ADDI R3,R3,1`]" Inv''' "R3" "R3" "1";;

th5;;

help_compose th5 th6;;

e(STRIP_TAC);;

e(ASM_REWRITE_TAC  [(MP (SPEC "R3 + 1" (SPEC "SUC R3"
                          (INST_TYPE [":num",":*"] EQ_SYM)))
               (SPEC "R3" ADD1))]);;
```

```
e(ONCE_REWRITE_TAC [EQ_SYM_EQ]);;

e(REWRITE_TAC [SUC_NOT]);;

e(REWRITE_TAC
         [LESS_OR_EQ;(REWRITE_RULE [SUC_NOT] (SPEC "SUC R3" LESS_0_CASES))]);;

let th6 = compose th5 th6 (top_thm());;

help_r_conseq th6 Inv;;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let th6 = r_conseq th6 (top_thm());;


help_compose th4 th6;;

e(STRIP_TAC THEN ASM_REWRITE_TAC []);;

let loop_thm = compose th4 th6 (top_thm());;

let rec_proc_thm = mk_while loop_thm;;

%-- the mk_send-s for the other processes will look like the following %

let th_A = mk_send "[` send mess_A to D `]" Inv "mess_A" "sigD";;

let th_B = mk_send Inv "mess_B" "sigD";;

let th_C = mk_send Inv "mess_C" "sigD";;
```

Bibliography

[1] G. R. Andrews. *Concurrent Programming : Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc. 1991.

[2] M. A. Arbib and S. Alagić. Proof Rules for **Goto**s *Acta Informatica*, 11, pages 139-148, 1979.

[3] W. R. Bevier. A Verified Operating System Kernel. Technical Report 11, Computational Logic, Inc., October 1987.

[4] W. R. Bevier. KIT: A Study in Operating System Verification. *IEEE Transactions on Software Engineering*, Vol. 15, No. 11, November 1989.

[5] R. Boyer and J. Moore. *A Computational Logic.* Academic Press, New York, USA, 1988.

[6] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs.* Elsevier Scientific Publishers, 1987.

[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[8] M. Clint and C. A. R. Hoare. Program Proving: Jumps and Functions. *Acta Informatica*, 1, pages 214-224, 1972.

[9] D. L. Clutterbuck and B. A. Carré. The Verification of Low-level Code. *Software Engineering Journal* , Vol. 3, No. 3, pages 97-111, May 1988.

[10] R. L. Constable. *Implementing Mathematics with the NUPRL Proof Development System.* Prentice Hall, 1986.

[11] Computer Science Laboratory, SRI International, Menlo Park, CA. EHDM Specification and Verification System Version 5.0 –Description of the EHDM Specification Language, January 1990.

[12] G. Fink, M. Archer, and L. Yang. PM: A Proof Manager for HOL and Other Provers. *Proceedings of the International Workshop on the HOL Proving System and Its Applications*, pages 286-304, 1991.

[13] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Lecture Notes in Computer Science No. 78. Springer Verlag, 1979.

[14] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 73–128. Kluwer Academic Press, 1988.

[15] M. Gordon. Mechanizing Programming Logics in Higher Order Logic. *Current trends in hardware verification and automated theorem proving.* G. Birtwistle, P.A. Subrahmanyam, editors. New York : Springer-Verlag, c1989.

[16] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12, 10(October 1969), pages 576-580, 583.

[17] C. A. R. Hoare. *Communicating Sequential Processes Communications of the ACM*, 21, 8(August 1978), pages 666-677.

[18] W. A. Hunt. FM8501: A Verified Microprocessor. Inst. Comput. Sci., Univ. Texas at Austin, Tech. Rep. 47, Dec. 1985. 576-580, 583.

[19] T. Kowaltowski. Axiomatic Approach to Side Effects and General Jumps. *Acta Informatica*, 7, pages 357-360, 1977.

[20] T. Melham. A Package for Inductive Relation Definitions in HOL. Technical Report, Cambridge University Computer Laboratory, 1989.

[21] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6, pages 319-340, 1976.

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System In the *Proceedings of the 11th International Conference on Automated Deduction*, 1992.

[23] J. M. Rushby and B. Randell. A Distributed Secure System. *IEEE Computer*, pages 55-67, July 1983.

[24] R. D. Schlichting and F. B. Schneider. Using Message Passing for Distributed Programming: Proof Rules and Disciplines. *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 3, pages 402–431, July 1984.

[25] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Wm. C. Brown Publishing Co., 1988.

[26] J. E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[27] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall Press, 1981.

[28] P. J. Windley. The Formal Verification of Generic Interpreters. Ph.D. dissertation, Dept. of Computer Science, University of California, Davis. June 1990.

@InProceedingsWirth:ldrs:1987, author = "N. Wirth", title = "Towards a Discipline of Real-Time Programming", crossref = "ldrs:1977", pages = "142", note = "Only the abstract of this paper appears in the conference proceedings since the paper was submitted for publication in Communications of the ACM.", checked = "19940409", abstract = "Programming is divided into three major categories with increasing complexity of reasoning in program validation: *sequential* programming, *multi*-programming, and *real-time* programming. By adhering to a strict programming *discipline* and using a suitable high-level *language* moded after this discipline, we may drastically reduce the complexity of reasoning about concurrency and execution time constraints. This may be the only practical way to make real-time systems analytically verifyable and ultimately reliable. A possible discipline is outline and expressed in terms of the language *Modula*.",