LCF LSM, A system for specifying and verifying hardware

Mike Gordon

September 1983
LCF_LSM
A system for specifying and verifying hardware

Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abstract

The LCF_LSM system is designed to show that it is practical to prove the correctness of real hardware. The system consists of a programming environment (LCF) and a specification language (LSM). The environment contains tools for manipulating and reasoning about the specifications. Verification consists in proving that a low-level (usually structural) description is behaviourally equivalent to a high-level functional description. Specifications can be fully hierarchical, and at any level devices can be specified either functionally or structurally.

As a first case study a simple microcoded computer has been verified. This proof is described in a companion report. In this we also illustrate the use of the system for other kinds of manipulation besides verification. For example, we show how to derive an implementation of a hard-wired controller from a microprogram and its decoding and sequencing logic. The derivation is done using machine checked inference; this ensures that the hard-wired controller is equivalent to the microcoded one. We also show how to code a microassembler. These examples illustrate our belief that LCF is a good environment for implementing a wide range of tools for manipulating hardware specifications.

This report has two aims: first, to give an overview of the ideas embodied in LCF_LSM, and second, to be a user manual for the system. No prior knowledge of LCF is assumed.

N.B. This is the second printing of Tech. Report No. 41. Various corrections and additions have been made.
Contents

Introduction ................................................................................................................. 1

Introduction to LCF ...................................................................................................... 2

The Syntax of LSM ........................................................................................................ 6

LSM Terms for Representing Sequential Machines .................................................... 12

Summary of LSM's Terms for Defining Devices ......................................................... 21

The Axioms and Inference Rules of LSM ................................................................. 22

Conclusions .................................................................................................................. 37

Future research ............................................................................................................. 37

Appendix 1: Types, terms and constants of LSM ...................................................... 39

Appendix 2: Axioms and Built-in Theorems ............................................................. 42

Appendix 3: Rules of Inference ................................................................................... 44

References .................................................................................................................... 50

Acknowledgements ..................................................................................................... 51
Introduction

The LCF_LSM system is designed to show that it is practical to prove the correctness of real hardware systems. As a first case study a simple microcoded computer has been successfully verified. This proof is described in a separate report [Gordon5]. Current research is aimed at applying LCF_LSM to examples supplied by industrial collaborators.

Although LCF_LSM was originally implemented to test ideas in verification, it can also be used for other activities requiring the manipulation of specifications, and for general programming. For example, in the companion report [Gordon5], we show how to derive from the microcode and associated control logic, an implementation of a hard-wired controller which is functionally equivalent to the microprogrammed controller. We also show how to use the system to code a microassembler. These examples illustrate our belief that LCF (which is the basis of LCF_LSM) is a good environment for implementing a wide range of tools for manipulating hardware specifications.

The name of the system is a concatenation of two acronyms:

LCF: Logic of Computable Functions. A computer system, designed and implemented by Robin Milner and his colleagues, for generating formal proofs interactively [Gordon et. al.].

LSM: Logic of Sequential Machines. A formal system which extends the logical calculus embedded in LCF with terms based on the behaviour expressions of Robin Milner's Calculus of Communicating Systems (CCS) [Milner].

LCF_LSM is implemented as an extension to Cambridge LCF, a descendant of Edinburgh LCF. The version of LCF_LSM described below is still experimental and so has a number of rough edges and inelegancies. It is, however, intended to be sufficiently robust and efficient for use on realistic problems.

Instead of the rather theoretical approach taken earlier (e.g. in [Gordon2]), I have recently been concentrating on practical applications and have left a number of mathematical questions pending. This does not mean that I regard theoretical issues as unimportant - indeed the current system would never have been conceived without the pioneering work of Milne, Milner, Plotkin and others. I hope that by studying the specification of relatively large real systems a new set of interesting theoretical questions will be generated, and so the next phase of work may well have to be mostly mathematical.

This report has two aims: first, to give an overview of the ideas embodied
in LCF-LSM, and second, to be a user manual for the system. I have
included sufficient description of LCF to enable readers unfamiliar with
it to understand the main ideas described here. Serious users of
LCF-LSM will, however, need to become familiar with more of LCF than is
covered. The book [Gordon et. al.], paper [Paulson1] and reports
[Paulson2, Paulson3] provide the necessary documentation. An
introductory paper is [Gordon4]. An early, non-mechanized, version of
LSM is described in [Gordon2], many of the examples described here
originated in these reports.

The kind of behaviour expressible in LSM can also be expressed in several
other languages; for example, MIDDLE [Dembinski & Budkowski],
temporal logic [Moszkowski] and the "synchronous logic" described in
[Ayres] as a source language for silicon compilation. Further comparison
of the various notations would be fruitful; perhaps LCF-LSM can be
combined with Ayres's techniques to yield a uniform framework for VLSI
verification and implementation? It would be nice to be able to prove
designs correct before compiling them to silicon.

Another system, with similar goals to LCF-LSM, is described in [Barrow].
This work is based on Prolog rather than LCF.

The approach described in this report seems most convincing when
registers, gates etc. are taken as primitive, rather than being defined in
terms of lower level technology dependent devices like transistors. The
low level behaviour realized by particular technologies (NMOS, CMOS
etc.) can be expressed in LSM, but the models studied so far are rather
crude [Gordon3]. More detailed behavioural representation at this level
is possible in Milne's calculus CIRCAL [Milne] and Hanna's predicate logic
based framework VERITAS [Hanna].

Introduction to LCF

The LCF system interfaces to the user via an interactive programming
language called ML (for Meta Language). When LCF is first run it will
output a prompt character #, the user can then input either an
expression (which will be immediately evaluated) or a declaration (which
will result in a variable being bound to a value, or a function being
defined). Below is an example session; lines typed by the user start with
#, all other lined are output by ML.

```
#2+3;;
5 : int

#let x = 2+3;;
z = 5 : int

#let f z = z+1;;
f = . : int -> int
```
\#f x; 
\> : int

First the user inputs the expression 2+3 followed by the terminator ;; and a carriage return. ML responds by printing the value of the expression and its type. Next the user inputs the declaration let x = 2+3, this causes the value of expression 2+3 to be bound to the variable x, ML indicates this effect by printing out the new value of x and its type. The user then inputs a function declaration let f x = x+1 which defines f to be the function that adds one to its argument. ML prints function values as - the type int \rightarrow int indicates that f requires an argument of type int (i.e. an integer), and that it also returns a result of type int. Finally the user inputs the expression f x which applies function f to argument x resulting in the integer 6.

let e, e1, e2, ... stand for arbitrary ML expressions, and x, x1, x2 for arbitrary ML identifiers (variables), then ML includes the following kinds of expressions:

()  
This constant expression (pronounced "empty") denotes the only value of type void. This value is typically returned by ML functions whose main effect is a side-effect (e.g. new_theory - see below).

0, 1, 2, etc.  
Evaluates to values of type int.

tuple, false  
Evaluates to values of type bool.

'<list of characters>'  
Evaluates to a value of type tok (for token). Tokens are ML's version of strings; they are often used to name things.

x  
Evaluates to the value currently bound to x. ML identifiers can be any sequence of letters, digits, primes ('') or underlines ('_') starting with a letter. Each ML identifier has an ML type (e.g. in the example session above x has type int).

e1 e2  
Function application: e1 must evaluate to a function, i.e. a value
with a type of the form \( ty1 \rightarrow ty2 \) (see the section below on types for the meaning of \( \rightarrow \)); and \( e2 \) to a value of type \( ty1 \). The application \( e1 \; e2 \) then evaluates to the result of applying the function denoted by \( e1 \) to the value of \( e2 \).

\[ e1 \; ix \; e2 \]

Here \( ix \) must be one of ML’s predefined binary operators. These include \(+\), \(-\), \(*\), \(/\), \(<\), \(>\), \(=\), \(or\), \(&\).

\( (e1, e2) \)

Evaluates to a pair whose first component is \( e1 \)'s value, and whose second component is \( e2 \)'s value. If \( e1 \) has type \( ty1 \) and \( e2 \) type \( ty2 \) then \( (e1, e2) \) has the product type \( ty1 \times ty2 \). The components of a pair can be extracted with the built-in ML functions \( fst \) and \( snd \). For example, \( fst(e1, e2) = e1 \) and \( snd(e1, e2) = e2 \).

\[ [e1; \ldots; en] \]

Evaluates to a list of the values of \( e1, \ldots, en \). Each \( ei \) must have the same type, \( ty \), say, and then the list \([e1; \ldots; en]\) has the type \( ty \) list. The standard list processing functions \( hd \), \( tl \), \( cons \) (which can be infixed as \( .\) ), \( null \) and the empty list \( nil \) are built-in. They satisfy:

\[ \begin{align*}
hd [e1:e2; \ldots; en] &= e1 \\
tl [e1:e2; \ldots; en] &= [e2; \ldots; en] \\
null nil &= true \\
null [e1; \ldots; en] &= false \\
e1. [e2; \ldots; en] &= [e1; e2; \ldots; en] \\
\end{align*} \]

\[ if \; e \; then \; e1 \; else \; e2 \]

The usual conditional: evaluates to the value of \( e1 \) if \( e \) evaluates to \( true \), and to the value of \( e2 \) otherwise. \( e1 \) must have type \( bool \), and \( e2 \) and \( e3 \) the same types.

\[ let \; d \; in \; e \]

This is a block with local declarations \( d \) (see below). The expression \( let \; d \; in \; e \) evaluates to the result of evaluating \( e \) in a local environment with bindings determined by \( d \). For example, \( let \; x=3 \; in \; x \times x \) evaluates to 9.

In addition to the kinds of expressions just described, ML also has expressions which evaluate to terms, types, formulae and theorems of a logical calculus - the object language (OL). In the LCF system, the object language is called PPLAMBDAA (which is an acronym derived from "Polymorphic Predicate Lambda-Calculus"). In the LCF-LSM system there is a different object language called LSM (Logic for Sequential
Machines) which has PPLAMBDA as a subset, but also contains some extra terms. In the descriptions that follow I will not be completely precise about which things are in PPLAMBDA and which are only in LSM - I will usually just refer to the logic as OL.

Constructs of OL have a special syntax, which must be surrounded by quotes when inputting to ML. For example "x+y" is an ML expression of type term (an OL term), and "x+y == y+x" is an ML expression of type form (an OL formula). Note that whereas 2+3 is an ML expression of type int, "2+3" is an ML expression of type term. The quotes separate the object language OL from the meta language ML. We shall describe OL shortly, but first we must say a little about ML declarations.

The following are the main kinds of declarations:

```ml
let x = e
```

This binds identifier x to the value of the expression e.

```ml
let x1, ..., xn = e
```

Here e must evaluate to a value of the from (v1, ..., vn), the declaration then simultaneously binds each xi to vi.

```ml
let f x = e
```

This defines f (which must be an ML identifier) to be the function with formal parameter x and body e.

```ml
let f x1 ... xn = e
```

This defines f to be a curried function of type ty1->...->tyn->ty where each xi has type tyi and e has type ty. If f is defined by this declaration and e1 has type ty1 then f e1 is an ML expression (called the partial application of f to e1) of type ty2->...->tyn->ty.

```ml
let (x1, ..., xn) = e
```

This defines f to be a function of type ty1#...#tyn->ty, i.e it takes a vector (tuple) as argument.

```ml
let b1 = e1 and b2 = e2 ... and bm = em
```

This simultaneously defines bi to be ei; bi can either be an identifier or something of the form f x1 ... xn. For example, let x = 1 and f y = x+y; in this the x in the body of f has whatever value x has before the declaration was executed (i.e. it is not necessarily i).
In the three kinds of function definitions described above the keyword 
let must be replaced by letrec if the function is recursive.

Let \( ty, ty1 \) and \( ty2 \) range over arbitrary ML types, then the types of ML 
include:

\[
\text{bool, int, tok, void, term, form, type, thm}
\]

These are predefined primitive types. The types \( \text{term, form, type} \) 
and \( \text{thm} \) are discussed in detail below.

\( ty1#ty2 \)

The type of ML pairs whose first component has type \( ty1 \) and 
second component has type \( ty2 \). (For example \( \text{true,3} \) has type 
\( \text{int}#\text{bool} \).)

\( ty \ l i s t \)

The type of ML lists of values of type \( ty \). (For example \( [1;2;3] \) has 
type \( \text{int list} \).

\( ty1->ty2 \)

The type of functions taking arguments of type \( ty1 \) and returning 
results of type \( ty2 \).

The Syntax of LSM

LSM differs from PPLAMBDA in two main ways: first, it has some extra 
kinds of terms loosely based on the behaviour expressions of CCS 
[Milner], and second, it does not contain the "undefined values" needed 
for fixed-point induction (Scott induction). Since I did not need induction 
for my first case study I simplified things by removing the associated 
paraphernalia (future studies might result in their reinstatement).

OL is interfaced to ML via four types:

\( \text{term} \) ML values of type \( \text{term} \) denote OL terms. Each such term has an OL 
type and denotes a value of that type. These values can be 
numbers, truthvalues, words, pairs, lists, functions, sequential 
machines etc. It is important to distinguish ML types and values 
from OL types and values. For example, \( 3 \) is an ML value with ML 
type \( \text{int} \), whereas "\( 3 \)" is an ML value of ML type \( \text{term} \), and this term
denotes an OL value with OL type $num$.

**type** ML values of type $type$ denote OL types. For example "$.num$" is an ML expression of type $type$ denoting the OL type $num$. The syntax of quoted OL types is "$.<type expression>$".

**form** ML values of type $form$ denote OL formulae. Such formulae are predicate calculus sentences. For example, "$orall t. t = T \lor t = F$" is an ML expression of type $form$ which denotes a OL formula that expresses the proposition that for every $t$ either $t$ is $T$ or $t$ is $F$.

**thm** ML values of type $thm$ are certain pairs $(fml,fm)$ where $fml$ is a list of formulae, and $fm$ is a formula. Such a pair is interpreted as asserting that $fm$ holds if all the formulae in $fml$ (called assumptions) hold. For example, for any $fm$ the assertion corresponding to $([fml], fm)$ always holds. The only way to construct values of type $thm$ is to use certain predefined ML procedures called inference rules. For example, the ML function $ASSUME : form \rightarrow thm$, when applied to any formula $fm$ generates the ML theorem represented by $([fml], fm)$. If $([fml1,...,fmln], fm)$ is a pair representing a value of ML type $thm$ we write $fml1,...,fmln \vdash fm$. The ML system normally prints such a theorem as ... $\vdash fm$ - each assumption is printed as a dot.

Note that terms, types and formulae can be directly input using the quotation syntax (details below), but theorems can only be created by the predefined inference rules. A tutorial introduction to ML and the concepts underlying the representation of OL in ML is [Gordon4]. I strongly suggest reading this if the brief remarks above are confusing.

Here is a session to illustrate the manipulation of OL from ML:

```ml
  # let fm = "\!t. t = T \lor t = F" ;
  fm = "\!t. t = T \lor t = F" : form

  # let th1 = ASSUME fm ;
  th1 = |- "\!t. t = T \lor t = F" : thm

  # let th2 = SPEC "x: bool" th1 ;
  th2 = . |- "x = T \lor x = F" : thm
```

First we bind $fm$ to a formula; then we apply the inference rule $ASSUME$ to $fm$ resulting in a theorem which we bind to the ML identifier $th1$. Then we use the inference rule $SPEC$ to specialize the quantified variable $t$ in $th1$ to $x$.

We now describe the parts of LSM which are similar (but not identical) to PPLAMBDA; for more details see the Manual [Paulson2]. In the next section we introduce the special terms which enable the behaviour and structure of hardware devices to be expressed.
When describing OL constructs we omit the surrounding quotes needed when inputting to ML (note that in the case of types these quotes are "": followed by ").

We use \( t, t_1, t_2 \) etc. to range over terms, and \( x, x_1, x_2 \) etc. over variables. The terms of OL are:

\[ 0, 1, 2, \text{etc.} \]

These have ML type \textit{term} and OL type \textit{num}.

\[ T, F \]

These have ML type \textit{term} and OL type \textit{bool}.

\[ \#b_1...b_n \]

Here each \( b_i \) is either \( 0 \) or \( 1 \). The term \( \#b_1...b_n \) has OL type \textit{word}n; for example, \( \#01101 \) has OL type \textit{word}5. OL has an infinite family of distinct types: \textit{word}1, \textit{word}2, etc. to represent bitstrings of different sizes. The type \textit{bool list} can be used to represent variable sized bitstrings. There is also an infinite family of types: \textit{tri_word}1, \textit{tri_word}2, etc. to represent tri-state values. These are needed for representing busses, and are explained in [Gordon5] in the context of the computer example described there.

\[ x \]

OL variables - any sequence of letters, numbers, primes (') or underlines (_) starting with a letter. Each variable has a OL type which determines the range of values it can take. To explicitly indicate this type one can write \( x:ty \) instead of just \( x \); in the absence of explicit type information the OL type-checker (like the ML one) tries to infer types from context.

\[ t_1 t_2 \]

Function application: \( t_1 \) must have a OL type of the form \( ty_1 \rightarrow ty_2 \), and \( t_2 \) type \( ty_1 \), then the application has type \( ty_2 \); it denotes the result of applying the function denoted by \( t_1 \) to argument \( t_2 \).

\[ t_1 i z t_2 \]

Here \( iz \) must be one of the standard OL binary operators (e.g. \( = \)); see Appendix 1 for details.

\( (t_1,t_2) \)
If \( t_1 \) and \( t_2 \) have OL types \( t_1y \) and \( t_2y \) respectively, then \((t_1,t_2)\) has OL type \( t_1\#t_2 \).

\([t_1;...,t_n]\)

Evaluates to an OL list of the values of \( t_1, ..., t_n \). Each \( t_i \) must have the same type, \( ty \) say, and then the list \([t_1;...,t_n]\) has the type \( ty \) list.

\((t \rightarrow t_1 | t_2)\)

This is the conditional "if \( t \) then \( t_1 \) else \( t_2 \). \( t \) must have OL type \( bool \), and \( t_1, t_2 \) the same OL type, \( ty \) say, the conditional term then has type \( ty \) also. (In PPLAMBDA the conditional has syntax \((t \Rightarrow t_1 | t_2)\) and \( t \) must have type \( tr \).)

\(let \ x=t_1 \ in \ t_2\)

This is equivalent to \( t_2 \) with all free occurrences of \( x \) replaced by \( t_1 \). For example, one can prove: \(let \ x=2 \ in \ x+x = 2*2\).

Besides numbers, truth-values and bits there are various built-in constant functions (a constant function is just a constant with a functional type, i.e. a type built using \( \rightarrow \)). For example, for each \( n \) there is a constant \( WORDn \) of OL type \( num \rightarrow wordn \) for converting a number to a bitstring (thus \( \vdash WORD5 13 = \#01101 \)). See Appendix 1 for a list of the built-in constants.

The user can introduce his own constants by creating a theory (see examples below for details).

OL types include:

\( bool \)

The type of OL truth-values \( T \) and \( F \).

\( num \)

The type of OL numbers \( 0, 1 \) etc.

\( wordn \)

An infinite family of types (one for each \( n \)) of bitstrings of various lengths. For example, \( word1, word2, word3 \) etc. are all types of LSM.
tri_wordn

An infinite family of types (one for each n) of tri-state bitstrings of various lengths. For example, tri_word1, tri_word2, tri_word3 etc. are all types of LSM. These tri-state types are used to represent the values on busses; there are special constants to represent floating (or high impedance) states (see Appendix 1).

memm\_n

An infinite family of types (one for each m and n) of memories with m-bit addresses and n-bit contents. For the computer example described in [Gordon5] the main memory has type mem13\_16 and the control store has type mem5\_30.

ty1#ty2

The type of pairs \((t1,t2)\) where \(t1\) has OL type \(ty1\) and \(t2\) OL type \(ty2\) respectively.

ty list

The type of lists of values of type \(ty\). (For example \([1;2;3]\) has OL type num list).

ty1->ty2

The type of functions taking arguments of type \(ty1\) and returning results of type \(ty2\).

The user can introduce his own types by creating a theory.

Note the potential for confusing ML and OL - beware!

The formulae of OL include the following \((fm, fm1, fm2 \ldots\) range over arbitrary formulae):

t1 == t2

 States that \(t1\) and \(t2\) denote the same value. (Note that there is also a term \(t1=t2\) of OL type bool; \((t1=t2)==T\) is equivalent to \(t1 == t2\))

\(P(t1,\ldots,tn)\)

Here \(P\) is a predicate constant (== is a built-in infixed predicate constant).
\[ x.fm \]

Read \( \forall \) as "for all" (universal quantification).

\[ ?x.fm \]

Read \( \exists \) as "there exists" (existential quantification).

\[ \sim fm \]

Not \( fm \) (negation)

\[ fm1 \land fm2 \]

\( fm1 \) and \( fm2 \) (conjunction).

\[ fm1 \lor fm2 \]

\( fm1 \) or \( fm2 \) (disjunction).

\[ fm1 \rightarrow fm2 \]

\( fm1 \) implies \( fm2 \)

\[ fm1 \leftrightarrow fm2 \]

\( fm1 \) if and only if \( fm2 \)

The theorems of \( OL \) are determined by the axioms and rules of inference. The axioms are a predefined set of values of type \( thm \); see Appendix 2 for a list of the current axioms. The rules of inference are predefined ML functions which return theorems as results. For example there is an axiom called \( BOOL\_CASES \) which is \( \vdash \mathit{if}\_\text{bool} t == T \lor t == F \) (\( BOOL\_CASES \) is a predefined ML identifier bound to this theorem); the function \( \text{ASSUME} \) described above is an example of a rule of inference. All the axioms and rules of inference currently implemented in the \( LSM \) version of \( OL \) are described in Appendix 3.

In order to introduce application dependent constants, types and axioms, one can set up a hierarchy of theories. Each such theory has a name, which is a token, together with a set of constants, types and axioms. Each theory may have zero or more parent theories; the constants, types and axioms of a theory’s parents (and parent’s parents etc.) are available in the theory. The \( LCF\_LSM \) system is initialised with a number of built-in constants, types and axioms. I shall refer to these as constituting the theory ‘\( lsm \)’, this should be thought of as acting like a
parent to any theories created by the user; it contains the types bool, num and wordn (for each n). For a complete description see Appendix 1.

LSM Terms for Representing Sequential Machines

The terms described above are essentially those of PPLAMBDA (except that PPLAMBDA has three truthvalues TT, FF and UU, and it doesn’t have numbers, words or lists built-in). LSM also contains some special terms (based on the behaviour expressions of CCS [Milner]) which we describe in detail below. It is these CCS-like terms which constitute the main difference between LCF and LCF_LSM.

Consider a device with input lines \( i_1, \ldots, i_m \) and output lines \( o_1, \ldots, o_n \):

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
```

Suppose this device also has some internal registers \( x_1, \ldots, x_p \), and that it behaves like a sequential machine as follows:

At each moment in time the values on the output lines are a function of the values in the registers (the state) and the values on the input lines.

The values in the registers stay constant until a clock pulse is received on the clock line. (Exactly how a clock pulse is realized physically is left unspecified - it could, for example, actually be two voltage level changes (two phases: phi1 and phi2), or a single pulse).

LSM contains special terms to represent such sequential machines. These terms have OL type dev (for "device").

To specify the behaviour of a machine one must:

Specify the value on each output line in terms of the values of the state registers and the values on the input lines.

Specify how the state changes when the device is clocked.

As an example consider a counter:
Here the input lines are \textit{switch} and \textit{in}, the only output line is \textit{out} and the only state variable is \textit{n}. The name of the device is \textit{COUNT}; we write \textit{COUNT(n)} to show that the behaviour (to be described) depends on \textit{n} (actually \textit{COUNT} would be an OL constant of type \textit{num->dev}). Suppose the behaviour of \textit{COUNT} is informally specified by:

The value on the output line \textit{out} is always the value of the state variable \textit{n}. We can express this with the output equation:

\[
\text{out} = n
\]

When the counter is clocked, the new value of the state variable \textit{n} becomes \texttt{n+1} (i.e. the old value plus one) if \texttt{false} is being input on line \textit{switch}, otherwise it becomes the value input on line \textit{in}. We can express this by:

\[
\text{CLOCK(n)} \rightarrow \text{CLOCK}\left(\text{switch} \rightarrow \text{in} \mid \text{n+1}\right)
\]

In LSM the behaviour of the counter would be specified by the formula

\[
\text{COUNT}(n) = \text{dev}\{\text{switch}, \text{in}, \text{out}\}, \{\text{out}=n\}; \text{COUNT}\left(\text{switch} \rightarrow \text{in}\mid \text{n+1}\right)
\]

This formula has the form \texttt{t1 == t2} where \texttt{t1} is the term \texttt{COUNT(n)} and \texttt{t2} is a new kind of term of type \texttt{dev} described below. Notice that the clocking is implicit in our notation (i.e. we don't explicitly mention the clock line). From now on we will not draw clock lines in diagrams, though they will still be needed in actual hardware implementations. Our model of behaviour abstracts away from the physical details of how state-changes are effected, and treats devices as abstract sequential machines. I hope the examples below show that this abstraction is justified - i.e. that significant aspects of correctness can still be expressed. The new terms for expressing sequential behaviour are:

\[
\text{dev}\{x1,\ldots,xm\},\{t1=t1,\ldots,tn=tn\}; t
\]

A term of this form is called a behaviour term. It denotes a sequential machine whose input and output lines are \texttt{x1}, \ldots, \texttt{xm}. If \texttt{t1} is not listed among \texttt{x1}, \ldots, \texttt{xm} then it is an internal (or local) line (these will be explained later). The term \texttt{t1} gives the value output on line \texttt{t1}. The new state after clocking is specified by the term \texttt{t}. Normally \texttt{t} will have the form \texttt{D(u1,\ldots,ur)} where \texttt{D} is a device name.
(e.g. \textsc{count}) and \(u_1, \ldots, u_r\) are terms giving the new values of the state variables of \(D\). If \(t_i\) occurs in \(t\) or one of the \(t_i\) then its value there is determined by the equations \([l_1=t_1, \ldots, l_n=t_n]\). The free variables of the whole behaviour term are the free variables in \(t_1, \ldots, t_n\) and \(t\) minus \(x_1, \ldots, x_m, l_1, \ldots, l_n\).

An example of a behaviour term is:

\[ \text{dev}(t, o).\{o=n\}; \text{reg}(t) \]

This specifies a device that outputs on line \(o\) the value of variable \(n\) (which is a free variable of the term) and then, when clocked, becomes a device with behaviour \(\text{reg}(t)\) - i.e. becomes the device \(\text{reg}\) in a state holding the value input in line \(t\). Suppose we specify \(\text{reg}\) to satisfy:

\[ \text{reg}(n) = \text{dev}(t, o).\{o=n\}; \text{reg}(t) \]

then this defines \(\text{reg}\) to be a device which always outputs its state, and stores the current value input - i.e. it delays by one clock cycle.

Formulae of this form - i.e. of the form:

\[ D(a_1, \ldots, a_p) = \text{dev}(x_1, \ldots, x_m).\{l_1=t_1, \ldots, l_n=t_n\}; D(u_1, \ldots, u_p) \]

are called behaviour equations. They are used to directly specify sequential machines. We will shortly show how to give a structural specification of a machine in LSM also.

Here is another example of a behaviour term:

\[ \text{dev}(\text{switch}, \text{in}, \text{out}).\{\text{out}=n\}; \text{count}(\text{switch} \rightarrow \text{in}|n+1) \]

This term also has no internal lines, and again \(n\) is the only free variable.

The counter informally described above can be specified by the behaviour equation:

\[ \text{count}(n) = \text{dev}(\text{switch}, \text{in}, \text{out}).\{\text{out}=n\}; \text{count}(\text{switch} \rightarrow \text{in}|n+1) \]

An example of a behaviour term with internal lines is:

\[ \text{dev}(\text{switch}, \text{in}, \text{out}).\{l_1=(\text{switch} \rightarrow \text{in}|l_2), \text{out}=n, l_2=\text{out}+1\}; \text{count}_\text{imp}(l_1) \]

Here \(l_1\) and \(l_2\) are internal lines; again the only free variable is \(n\).

If the device \(\text{count}_\text{imp}\) has a behaviour satisfying:

\[ \text{count}_\text{imp}(n) = \text{dev}(\text{switch}, \text{in}, \text{out}).\]

\[ \{l_1=(\text{switch} \rightarrow \text{in}|l_2), \text{out}=n, l_2=\text{out}+1\}; \text{count}_\text{imp}(l_1) \]
Then the rule of inference \textit{UNFOLD_IMP} to be described in Appendix 3 will enable us to "solve" the equations for the lines, and hence derive:

\[
\text{COUNT_IMP}(n) \equiv \text{dev\{switch, in, out\},}
\]
\[
l1= (\text{switch} \rightarrow \text{in}[n+1]), \text{out}=n, l2=n+1;
\]
\[
\text{COUNT_IMP}(\text{switch} \rightarrow \text{in}[n+1])
\]

Note that in this formula \(l1\) and \(l2\) are no longer used anywhere. The \textit{PRUNE_EQUATIONS} rule (described in Appendix 3) will enable the equations for these variables to be removed to get:

\[
\text{COUNT_IMP}(n) \equiv \text{dev\{switch, in, out\}.} [\text{out}=n]; \text{COUNT_IMP}(\text{switch} \rightarrow \text{in}[n+1])
\]

Note that this equation for \textit{COUNT_IMP} is similar to the equation specifying \textit{COUNT}. A rule called \textit{UNIQUENESS} will enable us to derive from this that:

\[
\text{COUNT}(n) \equiv \text{COUNT_IMP}(n)
\]

If the state of a device remains constant over time it is called combinational. Here are two examples:

\[
\text{switch } \textit{t1} \quad \textit{t2} \\
\text{MUX} \\
\text{out}=0
\]

\[
\text{4} \\
\text{INC} \\
\text{out}=0
\]

The value on the output line \(o\) of the multiplexor \textit{MUX} equals the value on line \(t1\) if the value on line \textit{switch} is \(T\), otherwise it is the value on line \(t2\). Thus the value on the output line is given by the output equation: \(o=(\text{switch}\rightarrow t1|t2)\). The behaviour of \textit{MUX} is thus specified by the behaviour equation:

\[
\text{MUX} \equiv \text{dev\{switch, t1, t2\}.} [o=(\text{switch}\rightarrow t1|t2)]; \text{MUX}
\]

Note the lack of state variables. The behaviour of \textit{INC} is specified by the behaviour equation:

\[
\text{INC} \equiv \text{dev\{t, o\}.} [o=(i+1)]; \text{INC}
\]

Thus \textit{INC} is a combinational device that always outputs (on line \(o\)) one plus the value input (on line \(t\)).

Behaviour terms are used to directly specify what devices are supposed to do. LSM can also be used to describe the structure of digital systems as the interconnection of separate devices. Consider the system below:
This device is built from components similar to \textit{MUX}, \textit{REG}(n) and \textit{INC} as described above, except that the line names have been changed. The multiplexer \textit{MUX}' is like \textit{MUX} except that it has lines \textit{in}, \textit{l2, l1} instead of \textit{i1}, \textit{i2}, \textit{o} respectively. The need to rename lines motivates the following kind of term:

\[ t \text{rn}[l_1=l_1';...;in=ln'] \]

Here \( t \) should be a term of OL type \textit{dev} and \( l_1, ..., in, l_1', ... ,ln' \) are line names (which must be OL variables). The term denotes a behaviour similar to that denoted by \( t \) except that each line \( li \) is systematically renamed to \( li' \).

Suppose that \textit{MUX} is as specified above, i.e. it satisfies:

\[ \text{MUX} \equiv \text{dev}[\text{switch}, i_1, i_2, o].\left[ o=(\text{switch}\rightarrow i_1\{i_2\}) \right]; \text{MUX} \]

Then if we specify \textit{MUX}' by the formula:

\[ \text{MUX}' \equiv \text{MUX} \text{rn}[l_1=in; i_2=i_2; o=l_1] \]

then it will follow (using the rule \textit{EXPAND-DEF} described in Appendix 3) that:

\[ \text{MUX}' \equiv \text{dev}[\text{switch}, in, l_2, l_1].\{l_1=(\text{switch}\rightarrow in\{l_2\})\}; \text{MUX}' \]

Note that line \textit{switch} has not been renamed. The register \textit{REG}' in the diagram above is defined by renaming the lines of the generic register \textit{REG} by:

\[ \text{REG}'(n) \equiv \text{REG}(n) \text{rn}[i=l_1; o=\text{out}] \]
Then we will be able to prove using \textit{EXPAND-DEF} that if \textit{REG} is defined as above, i.e.:

\[ \text{REG}(n) = \text{dev}\{i,o\}.\{o=n\};\text{REG}(i) \]

then:

\[ \text{REG}'(n) = \text{dev}\{i,\text{out}\}.\{\text{out}=n\};\text{REG}'(i) \]

Similarly we can define:

\[ \text{INC}' = \text{INC} r[n\{i=\text{out}; o=\text{i2}\} \]

and then prove:

\[ \text{INC}' = \text{dev}\{\text{out}, \text{i2}\}.\{\text{i2}=(\text{out}+1)\};\text{INC}' \]

Note that instead of defining \textit{MUX'}, \textit{REG'} and \textit{INC'} by renaming lines of \textit{MUX}, \textit{REG} and \textit{INC}, we could have defined them directly by the behaviour equations:

\begin{align*}
\text{MUX'} & \equiv \text{dev}\{\text{switch}, \text{in}, \text{i2}, \text{ll}1\}.\{\text{ll}1=(\text{switch} \rightarrow \text{in}\{\text{i2}\})\};\text{MUX'} \\
\text{REG}'(n) & \equiv \text{dev}\{\text{ll}, \text{out}\}.\{\text{out}=n\};\text{REG}'(i) \\
\text{INC}' & \equiv \text{dev}\{\text{out}, \text{i2}\}.\{\text{i2}=(\text{out}+1)\};\text{INC'}
\end{align*}

If we did things this way, then the above equations would be axioms rather than derived theorems. ICF-LSM gives one the option of either defining each component directly (using a behaviour equation), or as a renaming of a previously defined primitive.

To represent a schematic diagram, the first step is to define its components (either directly, or by renaming) so that lines which are to be connected have the same name. For example, in the diagram above we have arranged that output line of \textit{MUX'} is the same as the input line to \textit{REG'} (\textit{viz. ll}). The next step is to write down a term which denotes the result of connecting together the component devices. LSM has a special kind of term for this purpose:

\[ [\ll 1 | \ll 2 | \ldots | \ll n ] \]

Here each \ll i must be a term of OL type \textit{dev}. The term \[ [\ll 1 | \ldots | \ll n ] \]
then denotes the result of connecting together the \ll i's by joining lines with the same name. The lines of the resulting device are the union of the lines of each of the the component devices \ll i.

For example, if \textit{MUX'}, \textit{REG'} and \textit{INC'} are as above, then:

\[ [ [ \ll \text{MUX'} | \ll \text{REG'}(n) | \ll \text{INC'} ] ] \]

denotes the device with structure:
In this diagram the lines \( l1 \) and \( l2 \) are output lines. To represent the diagram in which these lines are internal we need another kind of term.

\[ t \text{ hide}\{l1, \ldots, ln\} \]

Here \( t \) must be a term of OL type \( \text{dev} \). Suppose it represents a system specified by a diagram with lines \( l1, \ldots, ln \), then \( t \text{ hide}\{l1, \ldots, ln\} \) represents the system specified by the same diagram except that lines \( l1, \ldots, ln \) are internalized.

For example, the diagram:

\[ \text{switch in} \]
\[ \text{MUX' } \]
\[ \text{REG'\{n\} } \]
\[ \text{INC' } \]
\[ \text{ll out } \]
\[ \text{i2 } \]

\[ \text{out} \]
Can be represented by the term:

\[
\begin{align*}
| & \text{MUX}' & | \text{REG}'(n) & | \text{INC}' & | \text{hide}[1,1,2] \\
\end{align*}
\]

One can also explain the effect of hiding in terms of behaviour equations. Suppose \textit{COUNT_IMP1} is like \textit{COUNT_IMP} (as described above) except that lines \texttt{l1} and \texttt{l2} are no longer internal, i.e.:

\[
\text{COUNT_IMP1}(n) \equiv \text{dev}[\text{switch}, \text{in}, \text{out}, \texttt{l1}, \texttt{l2}]. \\
\quad \{ \texttt{l1} = (\text{switch} \rightarrow \text{in}) \}, \texttt{l2} = n, \texttt{out} = \texttt{out} + 1; \\
\quad \text{COUNT_IMP1}(\texttt{l1})
\]

Then if we define:

\[
\text{COUNT_IMP2}(n) \equiv \text{COUNT_IMP1}(n) \text{ hide}[1,1,2]
\]

One can show (using inference rules described in Appendix 3) that:

\[
\text{COUNT_IMP2}(n) \equiv \text{dev}[\text{switch}, \text{in}, \text{out}]. \\
\quad \{ \texttt{l1} = (\text{switch} \rightarrow \text{in}) \}, \texttt{l2} = n, \texttt{out} = \texttt{out} + 1; \\
\quad \text{COUNT_IMP2}(\texttt{l1})
\]

Notice that in this behaviour equation lines \texttt{l1} and \texttt{l2} are internal, whereas in the equation for \textit{COUNT_IMP1} they are output lines.

It is not necessary to introduce the constants \textit{MUX}'\textit{'}, \textit{REG}'\textit{'} and \textit{INC}'\textit{'}. One can simply write:

\[
\begin{align*}
| & \text{MUX} & | \text{REG}(n) & | \text{INC} & | \text{hide}[1,1,2] \\
\end{align*}
\]

When diagraming such terms we will draw in explicit names to indicate how the lines of the devices have been renamed. For example, the term above will be drawn as:

```
switch in
    |--------------------------|
    | MUX                     |
    |                         |
    | l1                      |
    |--------------------------|
    | REG(n)                  |
    |                         |
    | INC                     |
    |--------------------------|
    | out                     |
```
Note that $MUX$, $REG$ and $INC$ (as defined by behaviour equations above) have different line names to those in this diagram. For example, $MUX$ was defined to have lines $i1$, $i2$ and $o$ instead of $in$, $l2$ and $l1$. This illustrates our convention of using the names in diagrams to indicate renaming.

The final kind of term in LSM is used when several "microcycles" of an implementation are used to achieve one "macrocycle" of a specification. For example, the specification of a processor might define a "virtual device" which takes one cycle to execute a machine instruction. An implementation might use several microinstructions (and hence several microcycles) to fetch, decode and execute a single machine instruction. To express the correctness of such an implementation one must somehow say that sequences of cycles in the implementation correspond to a single cycle in the specification. We describe an example like this in [Gordon5], but here we consider something a bit simpler:

\[
\begin{align*}
\downarrow & \quad \downarrow \\
\text{MULT_IMP}(m,n,t) & \\
\downarrow & \\
\text{done} & \quad o
\end{align*}
\]

We want to specify this device to have the following behaviour:

If

1. state variable $t$ is initially $T$ and
2. numbers $x$ and $y$ are held on lines $i1$ and $i2$ and
3. the device is clocked until a state is reached in which line $done$ has value $T$, then in this "done-state" the value on line $o$ will be the product $x \cdot y$

To express this in LSM we first define a "virtual device" $MULT$ which takes just one step to compute the product, and has no $done$ line:

\[
MULT(m) \equiv \text{dev}[i1,i2,o].\{o=m\};MULT(i1\cdot i2)
\]

We then use a new kind of term to define a behaviour in which sequences of steps of $MULT_IMP$ for which the $done$ line is $F$ are merged to a single step. We then assert this derived behaviour should equal the behaviour of $MULT$. This assertion is:

\[
MULT(m) \equiv \text{until done do MULT_IMP}(m,n,t)
\]

This has the form $t1 := t2$ where $t2$ is a new kind of term defined below:
until l do t

Here l must be an OL variable of OL type bool, and t a term of type
dev denoting a device with l as an output line. The term until l do t
then denotes a device with the same lines as t except l. One cycle
of this device consists of a sequence of cycles of t. To get a single
state transition of until l do t one repeatedly clocks t (holding
the inputs constant) until one reaches the first state in which line l
carries T.

The exact meaning of until l do t is best grasped from the rule UNTIL
(described below) and the examples of its use. For example, later we will
define MULT_IMP so that it has a behaviour satisfying:

\[
MULT_{-}\_IMP(m,n,t) = \ni\{ done; t; iL, i2, 0 \}
\ni\{ x = n, done = t \};
\ni\{ MULT_{-}\_IMP(t \rightarrow (i1=0 \rightarrow 0) \mid (i1=0 \rightarrow 0) + m),
(t \rightarrow \{i1 \mid n-1\}),
((t \rightarrow \{i1 \mid (n-1) \} = 0 \mid (i2=0)) \} \]

Under the assumption that 0-1 equals 0 we will use LCF_LSM to show
that:

\[
MULT(m) = \text{until done do } MULT_{-}\_IMP(m,n,l)\]

Summary of LSM's Terms for Defining Devices

Sequential Machine: \[ dev[x1, ..., xm].\{l1=t1, ..., ln=tn\}; t \]
Renaming: \[ t \text{ ren}[l1=l1'; ..., ln=ln'] \]
Joining: \[ [t \mid i1 \mid i2 \mid ... \mid tn] \]
Hiding: \[ t \text{ hide}[l1, ..., ln] \]
Merging Microcycles: \[ \text{ until } l \text{ do } t \]

In Appendix 1 we list various ML functions for manipulating (e.g.
constructing and destructing) these terms.
The Axioms and Inference Rules of LSM

LSM contains all of PPLAMBDA (see [Paulson2]) except those parts pertaining to the theory of complete partial orders. In particular LSM does not have the Scott induction rule, the constants $UU, TT, FF, FIX$ and $COND$, or the axioms for partial ordering (i.e. $<<$), monotonicity, minimality and fixed points. PPLAMBDA's three truth values $TT, FF$ and $UU$ of type $\tau$ are replaced in LSM by $T$ and $F$ of type $\textit{bool}$, and the conditional $COND$ is replaced by $SCOND$ of type $\textit{bool} \rightarrow \ast \rightarrow \ast \rightarrow \ast$ (where $\ast$ can be any type). Appendix 1 contains the various built-in axioms, together with their ML names.

The selection of rules currently included in LSM is rather ad hoc - I have just implemented what seemed needed for the examples I have done. It is hoped that future versions of the system will have a more complete and rationally chosen set. Some theoretical work is needed to isolate a minimal and independent collection of axioms characterizing the class of diagrams representable by terms (see [Milner] for an analysis of a different class of diagrams). Further experimental work is needed to develop a convenient suite of derived rules.

We will introduce various rules using the $COUNT$ example discussed above. We give below a session in which we verify that the specification:

$COUNT(n) == \texttt{dev\{switch, in, out\}; \{out=n\}; COUNT(switch->in|n+1)}$

is correctly implemented by the following device:

```
switch   in
     |    |    |
     |    |    | MUX
     |    |    | 11
     |    |    | 12
     |    |    |
     |    |    | REG(n)
     |    |    |    |
     |    |    |    | INC
     |    |    |    |    |
     |    |    |    |    | out
```

This can be represented in LSM by introducing a constant $\textit{COUNT-IMP}$ defined by:

22
COUNT_IMP(n) ==
  [ MUX  rn[i1=n; i2=12; o=11]
  REG(n) rn[i=11; o=out]
  INC  rn[i=out; o=12] ]
hide[11,12]

Where the primitives used in this implementation are specified by:

MUX    == dev[switch, i1, i2, o]. {o=(switch->i1|i2)}; MUX
REG(n) == dev[i, o]. {o=n}; REG(t)
INC    == dev[i, o]. {o=(i+1)}; INC

Formally we wish to prove that:

!n. COUNT(n) == COUNT_IMP(n)

The sessions that follow are intended to occur in sequence, so that one can
assume the effects of earlier sessions persist into later ones. To start
with, here is a session in which we create theory called COUNT
containing the specification of our counter; recall that everything
preceded by the prompt symbol # is typed by the user, and everything
else by the system.

@new_theory 'COUNT';
() : void
@new_constant ('COUNT', ":num->dev");
() : void
@new_axiom
@ ("COUNT", "COUNT(n) == dev[switch, in, out].
  \{out=n; COUNT(switch->in|n+1)\}\);
1 - "!n. COUNT n == dev[switch, in, out]. \{out=n; COUNT(switch -> in|n+1)\}"
  : thm
@close_theory();
() : void

To fully understand this and subsequent sessions, one needs to be
familiar with Cambridge LCF. We will try and provide enough
commentary so that if you don't know LCF you can still get the gist of
what is going on, but inevitably some things will seem unmotivated and
mysterious. The ML function new_theory has type tok->void; it creates a
new theory whose name is the token given as argument. The function
new_constant has type tok#type->void; it declares an OL constant with
the name and OL type passed to it. The function new_axiom has type
tok#form->thm; it takes a name and a formula and makes the formula
into an axiom with the given name, the resulting theorem is returned;
any free variables in the formula (e.g. n above) are automatically
universally quantified. Finally, the function close_theory of type void->void freezes the current theory; no new constants or axioms can
subsequently be added, all one can do is prove theorems and save them.

In the next session we create a theory named COUNT_IMP containing
the implementation of the counter, including the definition of the

23
primitives MUX, REG and INC. We use the ML function map which applies 
itself first argument (which must be a function) to each element of its 
second argument (which must be a list) and then returns the resulting 
list. Evaluating the apparently useless expression "switch:bool,n:nom,..." 
has the side effect of telling the OC type checker that switch has default 
OC type bool, n has default OC type num, etc. It must be admitted that a 
cleaner form of type declaration would be preferable. Exactly when 
explicit typing is needed in OC is fairly subtle. Beginners are advised to 
explicitly declare the type of each variable before it is used (after a while 
one learns when it is safe to allow OC types to be inferred from context; 
when a variable has a type it will keep it unless explicitly overwritten).

```ml

# new_theory 'COUNT_IMP';
()

map new_constant [("MUX",".:dev");("REG",".:num->dev");("INC",".:dev")];
[(());();()]; void list

"switch:bool,n:nom,i:nom,i1:nom,i2:nom,o:nom";
"switch,n,i,i1,i2,o" : term

map

# new_axiom
#
# "new_axiom
# [(MUX), "MUX 
# (REG), "REG(n) 
# (INC), "INC 
#
#

# "COUNT_IMP",
#
# "COUNT_IMP(n) == 
# [ MUX r\(i\!=\!in;i2\!=\!i2;o\!=\!o1\) ]
# [ REG(n) r\(i\!=\!i1;o\!=\!o2\) ]
# [ INC r\(i\!=\!i2;o\!=\!o2\) ]]
#
#

then

# close theory();
()

end

We now begin a session in which we will verify COUNT_IMP. First we 
create a new theory called COUNT_VER in which we will prove the 
desired theorem. This theory must have access to the theories COUNT 
and COUNT_IMP; this is achieved by declaring these theories to be 
parents of COUNT_VER using the ML function new_parent. The ML 
function axiom fetches an axiom from a theory; it takes the theory name 
and axiom name as parameters (in that order). After starting the new 
theory we retrieve the axioms COUNT and COUNT_IMP from their

24
respective theories and bind the resulting theorems to the ML names COUNT and COUNT_IMP. Perhaps confusingly we are using these names for three separate purposes: (1) as names of theories, (2) as names for certain axioms on these theories and (3) as ML names for the axioms. Rather than give each primitive a separate ML name it will be more convenient to gather their defining axioms into a list, called prims. To generate this list we use the function obtained by partially applying axiom to the token 'COUNT_IMP'; this yields a function of type tok->thm which when applied to a token, tn say, fetches the axiom named tn from the theory COUNT_IMP. This function is then mapped down a list of axiom names to get the corresponding list of axioms.

```haskell
let prims = map (axiom 'COUNT_IMP') ['MUX'; 'REG'; 'INC'];
```

We can now start to prove theorems. First we expand the definitions of the primitives in COUNT_IMP using the inference rule UNFOLD_IMP. This is an ML function of type thm list -> thm -> thm. It takes a list of device definitions and a theorem and replaces each instance of a left hand side of a definition occurring in the theorem by the corresponding right hand side. We bind the theorem resulting from the application of UNFOLD_IMP to the ML name th1. For a complete description of UNFOLD_IMP and all the other inference rules of LSM (except those inherited from PPLAMBDA) see Appendix 3.

```haskell
let th1 = UNFOLD_IMP prims COUNT_IMP;;
```

25
The next step is to perform line renaming. For example, we replace:

\[(\text{dev} \cdot \text{switch}, i1, i2, o) \cdot [o = (\text{switch} \cdot i1) \cdot (i2); \text{MUX}) \cdot \text{rn}[i1 = \text{in}; i2 = \text{out}; o = \text{in}] \]

by:

\[(\text{dev} \cdot \text{switch}, \text{in}, i2, i1) \cdot [i1 = (\text{switch} \cdot \text{in}) \cdot (i2); \text{MUX}) \cdot \text{rn}[i1 = \text{in}; i2 = \text{out}; o = \text{in}]]\]

This renaming is done using the inference rule \text{RENAME\_LINES}. We bind the resulting theorem to the ML name \(th2\).

\[\text{let th2 = RENAME\_LINES th1;}\]
\[\text{th2 =}\]
\[\text{let th3 = \text{COMBINE\_EQUATIONS th2;} th3 =}\]
\[\text{let th4 = \text{FOLD COUNT\_IMP th3;} th4 =}\]

The "next-state" part of the right hand side of \(th3\) can be simplified by folding in the definition of \text{COUNT\_IMP}. This is done using the inference rule \text{FOLD}.
Having got a single set of equations for the values on the lines, we can now solve them. We replace:

\[ l1 = (\text{switch} \to \text{in} \mid l2), \text{out} = n, l2 = (\text{out} + 1) \]

by:

\[ l1 = (\text{switch} \to \text{in} \mid n + 1), \text{out} = n, l2 = (n + 1) \]

The inference rule \texttt{UNWIND\_EQUATIONS} is used to do this. This rule has type \texttt{tok list -> thm -> thm}; the token list is a list of line names that one doesn't want unwound, an illustration of when this is useful is the line \texttt{bus} in the computer example in [Gordon5].

\begin{verbatim}
let th5 = UNWIND_EQUATIONS [] th4 in
  \{ "COUNT_IMP" n ==
    \{ switch, in, out \} : tm
      \{ l1 = (switch \to \text{in} \mid n + 1), out = n, l2 = (n + 1) \}
    \[ COUNT_IMP(switch \to \text{in} \mid n + 1) " COUNT_IMP(switch \to \text{in} \mid n + 1) " : thm
\end{verbatim}

Next we notice that the equations for lines \texttt{l1} and \texttt{l2} are not used anywhere. Since these lines are internal we can delete the equations for them. The inference rule \texttt{PRUNE\_EQUATIONS} is used to do this.

\begin{verbatim}
let th6 = PRUNE_EQUATIONS th5 in
  \{ "COUNT_IMP" n ==
    \{ switch, in, out \} : tm
      \{ out = n \}
    \[ COUNT_IMP(switch \to \text{in} \mid n + 1) " : thm
\end{verbatim}

We are now almost done. Notice that the theorem \texttt{th6} shows that \texttt{COUNT\_IMP} satisfies the same equation that was used to define the specification \texttt{COUNT}. The inference rule \texttt{UNIQUENESS} enables us to deduce that two devices have equal behaviour if they satisfy the same equation. This is valid because behaviour equations have unique solutions [Gordon1]. Using \texttt{UNIQUENESS} we prove the theorem expressing the correctness of \texttt{COUNT\_IMP} with respect to the specification \texttt{COUNT}. We then save the resulting theorem on the theory \texttt{COUNT\_VER} with name \texttt{CORRECTNESS}.

\begin{verbatim}
let th7 = UNIQUENESS COUNT th6 in
  \{ "COUNT" n == COUNT_IMP n " : thm

\$save_thm('CORRECTNESS', th7);;
\{ "\text{n. COUNT} n == COUNT_IMP n " : thm
\end{verbatim}

Instead of laboriously doing each of the above steps by hand, as above, we can define derived rules in ML which apply each inference rule in turn automatically. The rule \texttt{EXPAND\_IMP} below derives a behaviour equation from a structural description of an implementation, together with the definitions of the primitives used.
Using `EXPAND_IMP` (which is, in fact, built-in to LCF_LSM) we can define another derived rule `VERIFY` which does the whole correctness proof in one step.

```plaintext
\#let EXPAND_IMP l prims imp =
\#let th1 = UNFOLD_IMP prims imp
\#in
\#let th2 = RENAME_LINES th1
\#in
\#let th3 = COMBINE_EQUATIONS th2
\#in
\#let th4 = FOLD_imp th3
\#in
\#let th5 = UNWIND_EQUATIONS l th4
\#in
\# PRUNE_EQUATIONS th5;
EXPAND_IMP = - : (tok list -> thm list -> thm -> thm)

\#EXPAND_IMP [ ] prims COUNT_IMP;
\"COUNT_IMP n == dev\switch, in, out\. \{out=n\} \: COUNT_IMP\(\switch -> in \mid n + 1\)\" :
\end{quote}

Although the example just done is trivial, it does illustrate many of the things needed in bigger proofs. The computer example in [Gordon85] shows this.

Two things not illustrated by the `COUNT` example are the use of LCF theorem proving techniques to prove ancillary lemmas (verification conditions), and the inference rule needed to handle cycle merging when one has terms of the form `until l do t`. To illustrate these we now describe another example - a multiplier. First we will construct a system `MULT_IMP` with behaviour:

```plaintext
MULT_IMP(m, n, t) ==
    dev\{done, i1, i2, o\},
    \{o=m, done=t\};
    MULT_IMP(t -> (i1=0 -> 0 | i2) \mid (i1=0 -> 0 | i2+m),
        (t -> i1 | n+1),
        (t -> i1+1 \mid (n-1)-1)=0 OR (i2=0))
```

The we will show that:

```plaintext
\mid n. MULT(m) == until done do MULT_IMP(m, n, T)
```

where `MULT` is defined by:

```plaintext
MULT(m) == dev\{i1, i2, o\}, \{o=m\}; MULT(i1•i2)
```

Thus one "macrocycle" of `MULT` is implemented by a variable number of microcycles of `MULT_IMP`. (Note that we assume `0-1` is `0`; we could easily modify the implementation to avoid the need for this assumption).
The definition of \texttt{MULT\_IMP} we use is given by the diagram below (we leave the reader the exercise of working out the line names from the structural description in LSM which we give shortly).

Although this is a simple system, it is not instantly obvious that it correctly does multiplication. As with the previous example we create theories for the specification and implementation (named \texttt{MULT} and \texttt{MULT\_IMP} respectively).

\begin{verbatim}
#new_theory'MULT';
() : void

#new_constant('MULT', "\texttt{\textbf{mnu}} \rightarrow \texttt{\textbf{dnu}}");
() : void
\end{verbatim}
Before defining the implementation of the multiplier in LSM we make a theory containing the definitions of the various primitives used.

Next we define a theory MULT_IMP with primes as a parent. Note that we use a slightly more compact way of assigning default OL types to variables.
Now we can use the derived rule EXPAND_IMP defined above as the first stage of the verification of MULT_IMP.

```ml
# new_theory 'MULT_VER';
() : void

# map new.parent ['MULT'; 'MULT_IMP'];
Theory MULT loaded
Theory MULT_IMP loaded
[(); (); ] : void list

# save_thm
# ('MULT_IMP_EXPAND',
#  EXPAND_IMP
# [])
# (map
#   (azion 'prims')
#   [ ['MUX' ; 'REG'; 'FLIPFLOP'; 'DEC'; 'ADDER';
#     'ZERO_TEST'; 'ZERO'; 'OR_GATE'] )
```

31
\[ \text{axiom } \text{MULT_IMP} \land \text{MULT_IMP} \] ;

\[ \text{MULT_IMP}(m, n, t) = \]
\[ \text{dev} \]
\[ \{ \text{done}, o, i, 1, i2 \} \]
\[ \{ o = m, \text{done} = t \} ; \]
\[ \text{MULT_IMP} \]
\[ ((t \rightarrow (i1 = 0 \rightarrow 0 \mid i2) \mid (i1 = 0 \rightarrow 0 \mid i2) + m), \]
\[ (t \rightarrow i1 \mid n - 1), \]
\[ ((t \rightarrow i1 = 1 \mid (n - 1) - 1) = 0 \mid (i2 = 0)) \]

: thm

Notice that we have not yet closed the theory \text{MULT_VER}; we will want to add another axiom later. First must use the rule of inference called \text{UNTIL}. This represents the meaning of terms \text{until } l \text{ do } t \text{ which we described above. The general idea is this: suppose we have a behaviour equation:}

\[ D(a1, \ldots, ap) = \text{dev}[l, z1, \ldots, zm] ; \]
\[ [i1 = t1, \ldots, ln = tn]; \]
\[ D(u1, \ldots, up) \]

If we regard each step of \text{D} as a microcycle of \text{until } l \text{ do } D(a1, \ldots, ap), then the rule \text{UNTIL} allows us to derive a behaviour equation for cycles of the \text{until}-term. Ignoring some details (which are discussed below), the equation yielded by \text{UNTIL} is:

\[ \text{until } l \text{ do } D(a1, \ldots, ap) = \text{dev}[z1, \ldots, zm] ; \]
\[ [i1 = t1, \ldots, ln = tn]; \]
\[ \text{until } l \text{ do } D(f(u1, \ldots, up)) \]

where intuitively \text{f(u1, \ldots, up)} is the first state following \text{(a1, \ldots, ap)} in which line \text{l} has value \text{T}. Notice that the done-line \text{t} is not a line of the \text{until}-device.

The next-state function \text{f} must satisfy the following equation:

\[ f(a1, \ldots, ap) = (t \rightarrow (a1, \ldots, ap) \mid f(u1, \ldots, up)) \]

Here \text{a1, \ldots, ap} are the state variables of \text{D} and \text{u1, \ldots, up} are the expressions from the behaviour equation for \text{D} that specify the next state. The equation above uniquely defines \text{f}. The rule \text{UNTIL} generates this equation from the behaviour equation for \text{D}, and so constructs the definition of the next-state function for \text{until } l \text{ do } D(a1, \ldots, ap).

Note how the equation for \text{f} simply iterates the state transformation:

\[ (a1, \ldots, ap) \rightarrow (u1, \ldots, up) \]

until a state is reached in which the value on the output line \text{l} (which is given by \text{t}) is \text{T}. In other words \text{f(a1, \ldots, ap)} is the first state after \text{(a1, \ldots, ap)} in which the value of \text{t} is \text{T}.

The discussion just given is slightly oversimplified in that it ignores the
requirement that during sequences of microcycles which make up a macrocycle, the inputs are assumed constant. To reflect this we must make the inputs parameters to the next-state function $f$.

We define an input line of a behaviour term:

$$\text{dev}\{l, z_1, \ldots, z_m\}. \{l=t, l_1=t_1, \ldots, l_n=t_n\}; D(u_1, \ldots, u_p)$$

to be a line $z_i$ that occurs free in $t$, or in one of the $t_i$, or in one of the $u_i$. For example, the input lines of:

$$\text{dev}\{\text{done}, i_1, i_2, 0\}. \{\text{true}, \text{true} \Rightarrow t\}; \text{MULT}_\text{IMP}(t \Rightarrow (i_1=0 \Rightarrow n \Rightarrow i_2) | (i_1=0 \Rightarrow n \Rightarrow i_2) \Rightarrow n), \{t \Rightarrow i_1 \Rightarrow n \Rightarrow t\}, \{(t \Rightarrow i_1 \Rightarrow n \Rightarrow t) \Rightarrow 0 \Rightarrow (n \Rightarrow 0)\})$$

are $i_1$ and $i_2$.

To express the requirement that inputs are held constant during microcycles, we modify the definition of the next-state function for $\text{until } l \text{ do } D(a_1, \ldots, ap)$ by changing the definition of $f$ to:

$$f(z_1, \ldots, zq, a_1, \ldots, ap) = (t \Rightarrow (a_1, \ldots, ap) \mid f(z_1, \ldots, zq, u_1, \ldots, up))$$

where $z_1, \ldots, zq$ are the input lines. We call this equation the next-state equation for $\text{until } l \text{ do } D(a_1, \ldots, ap)$. Notice how this reflects the constancy of the input lines during microcycles.

We can now describe the LSM rule $\text{UNTIL}$. It takes a theorem of the form:

$$D(a_1, \ldots, ap) = \text{dev}\{l, z_1, \ldots, z_m\}. \{l=t, l_1=t_1, \ldots, l_n=t_n\}; D(u_1, \ldots, u_p)$$

and produces a theorem of the form:

$$f(z_1, \ldots, zq, a_1, \ldots, ap) = (t \Rightarrow (a_1, \ldots, ap) \mid f(z_1, \ldots, zq, u_1, \ldots, up))$$

$$\Rightarrow \text{until } l \text{ do } D(a_1, \ldots, ap) = \text{dev}\{z_1, \ldots, z_m\}. \{l_1=t_1, \ldots, l_n=t_n\}; \text{until } l \text{ do } D(f(z_1, \ldots, zq, u_1, \ldots, up))$$

This says that if $f$ satisfies the equation preceding the $\Rightarrow$ then $\text{until } l \text{ do } D(a_1, \ldots, ap)$ satisfies the behaviour equation following it. The $f$ here is a variable, though the equation it satisfies uniquely determines it. The user will usually introduce an OL constant for this uniquely defined function, $F$ say, so that he can instantiate $f$ to $F$ and apply modus ponens (the PPLAMBDA rule $\text{MP}$) to derive:

$$\text{until } l \text{ do } D(a_1, \ldots, ap) = \text{dev}\{z_1, \ldots, z_m\}. \{l_1=t_1, \ldots, l_n=t_n\}; \text{until } l \text{ do } D(F(z_1, \ldots, zq, u_1, \ldots, up))$$
Let us do this for our multiplier. First we must use the UNTIL rule. The user must give as parameters to UNTIL the names of the next-state function and the done-line. Thus the ML type of UNTIL is tok\(\rightarrow\)tok\(\rightarrow\)thm\(\rightarrow\)thm. The first argument token is the name of the next-state function. The second token is the name of the done-line. We will call the next-state function-variable for the multiplier mult_fn; the done-line is done.

```ml
# save_thm
# (∀MULT_IMP_UNTIL',
# UNTIL 'mult_fn' 'done' (theorem 'MULT_VER' 'MULT_IMP_EXPAND'));;

mult_fn
(i1, i2, m, n, t) =>
(t ->
(m, n, t) |
mult_fn
(i1, i2, m, n, t) =>
(t ->
(i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2 + m),
(t -> i1 | n - 1),
((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))

(m, n, t) =>
until done do MULT_IMP(m, n, t) =>
until

0, i1, i2.
0 = m:

until done do MULT_IMP
mult_fn
(i1, i2, m, n, t) =>
(t ->
(i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2 + m),
(t -> i1 | n - 1),
((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))

: thm
```

Notice how UNTIL produces a theorem of the form \(fm1 \Rightarrow fm2\), where \(fm1\) is the recursive definition of the next-state function (called mult_fn in the example above), and \(fm2\) is the behaviour equation of the until-termin (until done do MULT_IMP(m, n, t) in the example above).

Let us now introduce a constant MULT_FN defined by the next-state equation just generated. Since we have not yet closed theory MULT_VER we can include it there.

```ml
# new_constant(MULT_FN, "num\#num\#num\#num\#bool -> num\#num\#bool");
() : void

# new_axiom
# (∀MULT_FN',
# "i1 i2 m n t.
# MULT_FN(i1, i2, m, n, t) =>
# (t ->
# (m, n, t) |
# MULT_FN
# (i1, i2, m, n, t) =>
# (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2 + m),
# (t -> i1 | n - 1),
# ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0))
# );

MULT_FN(i1, i2, m, n, t) =>

```

34
(t ->
    (m, n, t) |
    MULT_FN
    (t1, t2, 
    (t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2 + m),
    (t -> i1 | n - 1),
    ((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))")
: thm

gclose theory();
(): void

We now need a lemma relating the recursively defined function MULT_FN to multiplication. The required lemma can be proved in LCF using mathematical induction and various arithmetical properties (including 0·1 = 0). We will not give the proof here, but just make the needed theorem an assumption as follows:

(let lemma =
ASSUME
"i1 i2,
MULT_FN(i1,i2,(i1=0->0|i2),i1,((i1-1)=0)OR(i2=0)) ==
(i1)*i2, ((i1-1)=0)OR(i2=0)->i1 | 1), T";

lemma =

"!i1 i2.
MULT_FN(i1,i2,(i1 = 0 -> 0 | i2),i1,((i1 - 1) = 0) OR (i2 = 0)) ==
i1 * i2,((i1 - 1) = 0) OR (i2 = 0) -> i1 | 1), T"
: thm

The rule ASSUME of ML type form->thm takes a formula fm to a theorem fm |- fm.

Next we specialize the variable mult_fn in MULT_IMP_UNTIL to the constant MULT_FN we have just introduced. Then finally we do modus ponens with the specialized theorem and the definition of MULT_FN.

(let th1 =
@ MP (SPEC "MULT_FN" MULT_IMP_UNTIL) (axiom 'MULT_VER' 'MULT_FN');
th1 =

"!m n t.
until done do MULT_IMP(m,n,t) ==
dev
|o,i1,i2|,
|o=m|;
until
done
do
MULT_IMP
MULT_FN
(i1, i2,
(t -> (i1 = 0 -> 0 | i2) | (i1 = 0 -> 0 | i2 + m),
(t -> i1 | n - 1),
((t -> i1 - 1 | (n - 1) - 1) = 0) OR (i2 = 0)))")
: thm

Now we specialize t to T and simplify the result using the OL axiom BOOL_COND_CLAUSES and lemma. The axiom BOOL_COND_CLAUSES is:

!z y. (T -> z | y) == z ∧ (F -> z | y) == y
The simplifier is a derived inference rule called \texttt{REWRITE\_RULE}; it takes a list of theorems and uses them to simplify a given theorem (see [Paulson1] for details).

\begin{verbatim}
let th2 = REWRITE\_RULE (BOOL\_COND\_CLAUSES; lemma) (SPEC "T" (SPEC "n" (SPEC "m" th1)));

th2 = "until done do MULT\_IMP(m,n,T) ==
  \begin{cases}
  0, \text{i1, i2},
  \text{i2=0};
  \text{until}
  \text{done}
  \text{do}
  \text{MULT\_IMP(i1 \ast i2, (((i1 - 1) = 0) OR (i2 = 0) \rightarrow (i1 | 1)), T)"
\end{verbatim}

The last step is to use the inference rule \texttt{UNIQUENESS} to prove that the specification \texttt{MULT} (which is an axiom on the theory \texttt{MULT} set up above) is correctly implemented by \texttt{MULT\_IMP}.

\begin{verbatim}
let th3 = UNIQUENESS (axiom 'MULT' 'MULT') th2;

th3 = "MULT m == until done do MULT\_IMP(m,n,T)" : thm
\end{verbatim}

Then we can save \texttt{th3} as the theorem \texttt{CORRECTNESS}.

\begin{verbatim}
save_thm('CORRECTNESS', th3);

((i1 \ast i2),
MULT\_FN(i1,i2,(i1 = 0 \rightarrow 0 | i2),i1, (((i1 - 1) = 0) OR (i2 = 0)) ==
  i1 \ast i2, (((i1 - 1) = 0) OR (i2 = 0) \rightarrow (i1 | 1), T) \implies
MULT m == until done do MULT\_IMP(m,n,T)"
\end{verbatim}

When a theorem is saved in a theory all its assumptions are automatically discharged. Hence in the example above, the assumed lemma appears as the antecedent of an implication whose consequent is the formula we have just proved. If we subsequently prove this lemma as a theorem, we can remove it as an antecedent using modus ponens. We shall not do this here as it is routine LCF.

This concludes the correctness proof of the multiplier example, and also our introduction to LCF\_LSM. Although the examples we have used are trivial, I hope they are suggestive of what might be done. A much bigger example is given in [Gordon5].
Conclusions

Is it practical to prove real hardware correct?

The LCF_LSM verification system has been built to try and answer this question.

Based on the case studies done so far (primarily [Gordon5]), I claim that:

1. Non-trivial hardware can be proved correct using existing techniques.
2. Machine assistance is essential for such proofs.

Further evidence of this is provided by the work of John Herbert, a research student in the Computer Laboratory. He has proved correct one of the integrated circuits used in the Cambridge Fast Ring (a 100 megabit per second local network). The chip that has been verified translates between a serial data path and a sequence of 8-bit packets; it is implemented in ECL technology.

Future research

It is hoped to apply LCF_LSM (or its descendants) to examples supplied by industrial collaborators. I am actively seeking partners for this enterprise.

An industrially supported student (Inder Dhingra) is developing techniques specialized to the cMOS technology.

I am currently simplifying LCF_LSM using ideas from Temporal Logic. This work is being done in collaboration with my colleague Dr. Ben Moszkowski (who has a research position in the Computer Laboratory). It is also inspired by suggestions made by Edmund Ronald and Professor Tony Hoare. The idea is to dump LSM and just work with LCF. For example, instead of describing the counter specification and implementation with functions COUNT and COUNT_IMP, where:

\[
\text{COUNT}(n) \equiv \text{dev[switch.in.in; out=n]; COUNT(switch \rightarrow \text{in} \mid n+1)}
\]

\[
\text{COUNT_IMP}(n) \equiv \begin{cases} 
\text{MUX} & \text{rn}[t1=\text{in}; t2=\text{i2}; o=\text{i1}] \\
\text{REG}(n) & \text{rn}[t1=\text{i1}; o=\text{out}] \\
\text{INC} & \text{rn}[t2=\text{out}; o=\text{i2}] \\
\text{hide}[t1, t2] 
\end{cases}
\]

we use predicates COUNT_PRED and COUNT_IMP_PRED defined by:
\[
\text{COUNT\_PRED}(n, i, \text{switch}, o) \iff (o == n) \land \text{DEL}(n, (\text{switch} \rightarrow i \mid n + 1))
\]

\[
\text{COUNT\_IMP\_PRED}(n, i, \text{switch}, o) \iff \langle i \mid 1 \mid 2, \\
\quad \text{MUX\_PRED}(\text{switch}, i, 11, 12) \land \\
\quad \text{REG\_PRED}(n, 11, o) \land \\
\quad \text{INC\_PRED}(o, 12) \rangle
\]

where the primitives are specified by:

\[
\text{MUX\_PRED}(\text{switch}, i, 11, 12, o) \iff o == (\text{switch} \rightarrow i \mid 11 \mid 12)
\]

\[
\text{REG\_PRED}(n, i, o) \iff (o == n) \land \text{DEL}(n, i)
\]

\[
\text{INC\_PRED}(i, o) \iff o == i + 1
\]

This approach appears to have the following nice properties:

1. Formal descriptions are in "pure logic"; the extra CCS-like terms are not needed. For example, hiding is represented by existential quantification and parallel composition (i.e. [ ] ... [ ] ... [ ]) by conjunction.

2. Verification can be done using standard inference rules of logic. The \textit{ad-hoc} rules in Appendix 3 are not needed.

3. Combinational devices are not forced to have a next-state part. Sequential behaviour is specified with delay operators. (The predicate \text{DEL} is the unit-delay of temporal logic [Moszkowski].)

4. Bidirectional devices have a more natural specification.

5. Devices can be specified to have "pure delay" instead of state. This turned out to be useful for the ECL chip verification mentioned above.

Various case studies are being done to see if these nice properties really hold. For example, the cMOS work is being conducted in this framework.
Appendix 1: Types, terms and constants of LSM

LSM has the following types of arity 0 built in: \texttt{bool} (booleans), \texttt{num} (non-negative integers), \texttt{wordn} (n-bit words), \texttt{tr-wordn} (n-bit tri-state words), \texttt{memm}_n (m by n memories) and \texttt{dev} (devices). The type \texttt{list} of arity 1 is also built in.

The current implementation of LCF-LSM requires the user to explicitly declare the sizes of words and memories that he will use. Two ML functions are provided for this purpose:

\begin{verbatim}
declare_word_widths : int list -> void
declare_memories : (int * int) list -> void
\end{verbatim}

In the computer example described in [Gordon3], words of sizes 2, 3, 5, 13, 16 and 30, and memories of sizes 5 by 30 and 13 by 16 are used. The declarations needed for this are:

\begin{verbatim}
declare_word_widths[2;3;5;13;16;30]
declare_memories[(5,30);(13,16)]
\end{verbatim}

These declarations call \texttt{new\_type} for the appropriate types of the form \texttt{wordn}, \texttt{tr-wordn} and \texttt{memm}_n. They should be done in a theory that is a parent to all the theories that use \texttt{word} and \texttt{mem} types. The theory \texttt{values} has this role in the computer example.

In addition to the usual PPLAMBDA terms, LSM also has the following terms of OL type \texttt{dev}:

Sequential Machine: \hspace{1cm} \texttt{dev[x1,\ldots,xm].\{l1=t1,\ldots,ln=tn\};t}

Renaming: \hspace{1cm} \texttt{t rn[l1=t1;\ldots;ln=tn]}

Joining: \hspace{1cm} \texttt{\[t1\mid t2\mid\ldots\mid tn\]}

Hiding: \hspace{1cm} \texttt{t hide[l1,\ldots,ln]}

Merging Microcycles: \hspace{1cm} \texttt{until t do t}

Here \texttt{x1}, \ldots, \texttt{xm}, \texttt{l1}, \ldots, \texttt{tn} are OL variables used to represent lines.

For each of these terms there are ML syntax functions with prefixes \texttt{mk\_}, \texttt{is\_} and \texttt{dest\_} as described in [Paulson2]. The constructors and destructors are inverses; we just describe the former.

\texttt{mk\_dev(["x1";\ldots;"xm"], ["l1","t1";\ldots;"ln","tn"], "t")}

\texttt{"dev[x1,\ldots,xm].\{l1=t1,\ldots,ln=tn\};t"}
An ML-like syntax for lists is also available in LSM: If $t_1, \ldots, t_n$ all have OL type $ty$, then the term $[t_1; \ldots; t_n]$ has OL type $ty$ list. The empty list of OL type $* list$ is denoted by $[]$.

LSM also has terms: let $x = t_1$ in $t_2$. These are just alternative syntax for: $(\lambda x.t_2)t_1$ (where $\lambda x.t_2$ is the PPLAMBDA notation for lambda-expressions).

The LSM syntax for conditionals is $(t \Rightarrow t_1 | t_2)$ where $t$ is a term with OL type $bool$ and $t_1$ and $t_2$ have the same OL type. Note that this differs from the PPLAMBDA (and ML) syntax of $(t \Rightarrow t_1 | t_2)$ where $t$ must have type $tr$ (the PPLAMBDA type of three-valued truthvalues).

The built-in (non-functional) constants of LSM are:

- $T, F$ The truth-values of OL type $bool$
- $0, 1, 2, \ldots$ Numbers of OL type $num$
- $#b_1 \ldots b_n$ Words of OL type $wordn$ ($b_i$ is either 0 or 1).
- $[]$ The empty list of OL type $*list$.

The following infixed binary operators are built in:

- $=$ equality. OL type $* = bool$
- $+$ addition. OL type $num + num$->num
- $-$ subtraction. OL type $num - num$->num
- $*$ multiplication. OL type $num * num$->num
- $OR$ disjunction. OL type $bool OR bool$->bool
- $AND$ conjunction. OL type $bool AND bool$->bool
- $NOR$ negated disjunction. OL type $bool NOR bool$->bool
- $XOR$ exclusive OR. OL type $bool XOR bool$->bool
- $EQV$ logical equivalence. OL type $bool EQV bool$->bool

In addition there are the following (non-infixed) functions:
successor function. OL type num->num
predecessor function. OL type num->num
negation. OL type bool->bool
list cons. OL type *-> *list-> *list
head. OL type *list-> *
tail. OL type *list-> *list
null test. OL type *list-> bool
nth element of a list. OL type num-> *list-> *
sublist of a list. OL type num#num-> *list-> *list
number denoted by a bit list. OL type bool list-> num

The ML function declare_word_widths creates the following functions for each width declared:

VALn number denoted by a word. OL type wordn->num
WORDn word representing a number. OL type num->wordn
BITSn list of bits in a word. OL type wordn->bool list
NOTn complement a word. OL type wordn->wordn
MK_TRIn make a tri-state value.
        OL type wordn->tri_wordn
DEST_TRIn convert a tri-state value to a word.
        OL type tri_wordn->wordn

For each n there is also a constant FLOATn of OL type tri_wordn to represent the value on a floating bus of width n. The following infixes are also defined for each width:

ORNn bit-by-bit OR. OL type wordn#wordn->wordn
ANDn bit-by-bit AND. OL type wordn#wordn->wordn
Un combining tri-state values.
        OL type tri_wordn#tri_wordn->tri_wordn

The ML function declare_memories creates the following functions for each memory size (m,n):

STOREm store a value.
        OL type wordm->wordn->memm_n->memm_n
FETCHm fetch a value.
        OL type memm_n->wordm->wordn

Note that only one type of memory with a given address size is possible. For example, one can't have both mem13_8 and mem13_16. This restriction is purely to keep the names of the fetch and store functions short; it may be relaxed in the future.
Appendix 2: Axioms and Built-in Theorems

Little attempt has been made to give a well rounded set of axioms and built-in theorems. The ones listed below are motivated by the examples that have been done (notably the computer example in [Gordon4]). I expect that future studies will expose the need for more.

The following axioms are included in LSM. We give their ML name followed by the corresponding formula.

\[ 
\text{EQ} \quad \neg x :^* \land \neg y :^* \land z = y \Rightarrow \neg x = y \land (T \Rightarrow z \land y) = x \land (F \Rightarrow z \land y) = y \\
\text{BOOL_COND_CLAUSES} \quad \neg x :^* \land \neg y :^* \land (T \Rightarrow z \land y) = x \land (F \Rightarrow z \land y) = y \\
\text{BOOL_CASES} \quad \neg b : \text{bool}, b = \neg b \Rightarrow b = b \\
\text{BOOL_EQ_DISTINCT} \quad \neg T = \neg F \Rightarrow \neg F = \neg T \\
\text{NOT} \quad \neg \neg F = T \land \neg \neg T = F \\
\text{OR} \quad F \lor \neg \neg F = F \land F \lor \neg \neg F = F \land T \lor \neg \neg F = \neg \neg T \land T \lor \neg \neg F = \neg \neg T \\
\text{AND} \quad \neg b_1 \land \neg b_2 : \text{AND} \land b_2 = \neg \neg ((\neg \neg b_1 \land \neg \neg b_2)) \\
\text{NOR} \quad \neg b_1 \land \neg b_2 : \text{NOR} \land b_2 = \neg \neg ((\neg \neg b_1 \land \neg \neg b_2)) \\
\text{XOR} \quad \neg b_1 \land \neg b_2 : \text{XOR} \land b_2 = (b_1 \lor \neg \neg b_2) \land (\neg \neg b_1 \land \neg \neg b_2) \\
\text{EQV} \quad \neg b_1 \land \neg b_2 : \text{EQV} \land b_2 = (b_1 \lor \neg \neg b_2) \land (\neg \neg b_1 \land \neg \neg b_2)
\]

The following consequences of these axioms are available:

\[ 
\text{NEG_F} \quad \neg x \land \neg y : \text{NEG} \land x = y \Rightarrow \neg x = y \land \neg x = y \\
\text{F_NEG} \quad \neg x \land y = z \Rightarrow \neg x = y \land \neg x = y \\
\text{NEG_EQ} \quad \neg x : \text{NEG} \land \neg y : \text{EQ} \land x = y \Rightarrow \neg x = y \\
\text{EQ_T} \quad \neg x : \text{EQ} \land \neg y : \text{T} \Rightarrow \neg x = y \\
\text{NEQ_T_F} \quad \neg x : \text{NEQ} \land \neg y : \text{T} \Rightarrow \neg x = y \\
\text{F_T} \quad \neg x : \text{F} \land \neg y : \text{T} \Rightarrow \neg x = y \\
\text{NEG_F_T} \quad \neg x : \text{NEG} \land \neg y : \text{T} \Rightarrow \neg x = y \\
\text{T_F} \quad \neg x : \text{T} \land \neg y : \text{F} \Rightarrow \neg x = y \\
\text{BOOL_EQ} \quad \neg x : \text{BOOL_EQ} \land \neg y : \text{EQ} \land x = y \Rightarrow \neg x = y \\
\text{FN_COND} \quad \neg x : \text{FN_COND} \land \neg y : \text{COND} \land y = z \Rightarrow \neg x = y \\
\text{TRIV_COND} \quad \neg x : \text{TRIV_COND} \land \neg y : \text{COND} \land y = z \Rightarrow \neg x = y \\
\text{COND_PAIR} \quad \neg x : \text{COND_PAIR} \land \neg y : \text{PAIR} \land y = z \Rightarrow \neg x = y \\
\]

42
AND_TABLE
F AND F == F ∧
F AND T == F ∧
T AND F == F ∧
T AND T == T
NOR_TABLE
F NOR F == T ∧
F NOR T == F ∧
T NOR F == F ∧
T NOR T == F
XOR_TABLE
F XOR F == F ∧
F XOR T == T ∧
T XOR F == T ∧
T XOR T == F
EQV_TABLE
F EQV F == T ∧
F EQV T == F ∧
T EQV F == F ∧
T EQV T == T
OR_CLAUSES
t.  t OR T == T ∧
    T OR t == T ∧
    t OR F == t ∧
F OR t == t
AND_CLAUSES
    t.  t AND T == t ∧
    T AND t == t ∧
    t AND F == F ∧
    F AND t == F
DEMORGAN_OR
    t1 t2. NOT(t1 OR t2) == NOT t1 AND NOT t2
DEMORGAN_AND
    t1 t2. NOT(t1 AND t2) == NOT t1 OR NOT t2
NOT_NOT
    t.  NOT(NOT t) == t

At present there are no built-in axioms or theorems for numbers, words, memories or lists. In Appendix 3 we describe some inference rules which enable certain constant expressions at these types to be simplified (for example 2+3 can be simplified to 5).
Appendix 3: Rules of Inference

LSM contains inference rules for reasoning about the various terms of type \texttt{dev}. We give the ML type of each rule followed by a schematic description of its effect.

\textbf{COMPOSE} : \texttt{term -> thm}

\begin{verbatim}
COMPOSE
 "[| \texttt{dev \textit{X}1.EQ1;N1 \ldots \texttt{dev \textit{X}n.EQn;Nn}} | \texttt{hide \textit{L}}]
 \rightarrow
 "[| \texttt{dev \textit{X}1.EQ1,N1 \ldots \texttt{dev \textit{X}n.EQn,Nn}} | \texttt{hide \textit{L} = =
 \texttt{dev (X1u \ldots uXn)L.EQ1u \ldots uEQn;\{[| \texttt{N1 \ldots \texttt{Nn}}\}| \texttt{hide \textit{L}})}]
\end{verbatim}

Here \textit{X1u} \ldots \textit{uXn} and \textit{EQ1u} \ldots \textit{uEQn} denotes the union of sets of lines \textit{X1}, \ldots, \textit{Xn} and equations \textit{EQ1}, \ldots, \textit{EQn} respectively; \((\textit{X1u} \ldots \textit{uXn})\setminus\textit{L}\) denotes the union of the lines minus the lines in \textit{L} (thus \(-\) is set subtraction, or complement).

\textbf{COMPOSE} fails if:

1. \textit{L} contains an \texttt{l} which is an input line (i.e. occurs in an \texttt{NI} or the rhs of an equation in an \texttt{EQ} and is not the lhs of an equation).
2. There exist distinct \texttt{i} and \texttt{j} such that \textit{EQi} contains \texttt{x=ti} and \textit{EQj} contains \texttt{x=tj}.

\textbf{UNFOLD_IMP} : \texttt{thm list -> thm -> thm}

\begin{verbatim}
UNFOLD_IMP
 "[| \texttt{t1=ui} \ldots ;| \texttt{tm=um} |]
 \rightarrow
 "[| \texttt{t = = \{}\texttt{t1'} \texttt{rn[R1]} \ldots \texttt{tn'} \texttt{rn[Fn]}\}| \texttt{hide \textit{L}}]
\end{verbatim}

Where if \texttt{ti} has an instance \texttt{ti'} then \texttt{ui'} is the corresponding instance of \texttt{ui} (i.e. \texttt{ui'} is got from \texttt{ui} using the same substitution that yields \texttt{ti'} when applied to \texttt{ti}).

\textbf{UNFOLD_DEF} : \texttt{thm list -> thm -> thm}

\begin{verbatim}
UNFOLD_DEF
 "[| \texttt{t1=ui} \ldots ;| \texttt{tm=um} |]
 \rightarrow
 "[| \texttt{t = = \{}\texttt{ti'} \texttt{rn[R]}\}|"
\end{verbatim}

Where, as above, \texttt{ui'} is the instance of \texttt{ui} obtained by matching \texttt{ti'} to \texttt{ti} ("\texttt{|ti=ui}"") must not have any assumptions.

44
FOLD : thm -> thm -> thm

FOLD
  "|- t == u"
  "|- tm == devX,EQS,u"
  "|- tm == devX,EQS,t"

Where t' is the instance of t corresponding to the way u' is an instance of u.

RENAME : term -> thm

RENAME
  "(devX,EQS,NXT)rn[R]"
  "|- (devX,EQS,NXT)rn[R] == devX'.EQS';(NXT' rn[R])"

Where X', EQS' and NXT' are derived from X, EQS and NXT respectively by renaming according to R. RENAME fails if:

1. R has the form [...x=a;...y=a;...].
2. "x:ty1" is in X and "x:ty2" is a lhs of an equation in R and ty1, ty2 are distinct.

UNWIND : tok list -> term -> thm

UNWIND
  L
  "devX.[o1=t1, ..., on=tn]:NXT"
  "|- devX.[o1=t1, ..., on=tn]:NXT == devX.[o1=t1', ..., on=tn']:NXT'"

Where t1', ..., tn' and D' are got from t1, ..., tn and D by unwinding those equations whose lhs is not in L.

PRUNE : term -> thm

PRUNE
  "devX.EQS;NXT"
  "|- devX.EQS;NXT == devX.EQS';NXT"

Where EQS' is the subset of EQS obtained by removing equations whose lhs (i) is not an output line, and (ii) does not occur in D or the rhs of an equation used to define variables whose values are observable.

UNIQUENESS : thm -> thm -> thm

UNIQUENESS
  "|- D1(a1, ..., ap) == devX1.EQ1;D(A1, ..., Ap)"
  "|- D2(b1, ..., bq) == devX2.EQ2;D(B1, ..., Bq)"
  "|- D1(a1, ..., ap) == D2(b1, ..., bq)"

Where:
1. For each $i$: if $ai$ is not a variable then $ai$ equals $Ai$ and if $bi$ is not a variable then $bi$ equals $Bi$.
2. $X1$ denotes the same set of lines as $X2$, and $EQ1$ the same set of equations as $EQ2$.
3. The subset of pairs $(ai, Ai)$ where $ai$ is a variable occurring free in either the right hand side of an equation in $EQ1$ or in an $Ai$, equals the subset of pairs $(bi, Bi)$ where $bi$ is a variable occurring free in either the right hand side of an equation in $EQ2$ or in a $Bi$.

The easiest way to understand these conditions is to look back at the $COUNT$ and $MULT$ examples and see what they mean there.

$$\text{UNTIL : tok -> tok -> thm -> thm}$$

$$\text{UNTIL}$$

\[f\]

\[\text{let } \text{lp=}\text{up in}
\]

\[\text{until } \text{l do } \text{D}(\text{x1},...,\text{zn}) = \text{devx}.\text{EQS}\cdot\text{D}(\text{l1},...,\text{lp=}\text{up});\text{D}(\text{E1},...,\text{Ep})\]

Where $t1,...,tr$ are the input lines occurring in $t,E1,...,Ep,u1,...,up$ and $l1,...,lp$ are the internal lines (i.e. bi is not in X). $\text{UNTIL}$ fails if:

1. $l$ is not in $X$, or $l$ does not have type $\text{bool}$,
2. $x1,...,xn$ are not distinct, or some $xi$ is not a variable,
3. one of $E1,...,Ep$ has a free variable which isn't in $X$ or
4. any of $u1,...,up$ have free variables which are in $X$ or $\{x1,...,xn\}$

(thus the local equations $\{l1=u1,...,lp=up\}$ must not be recursive).

The rules that follow are not primitive (i.e. there are definable in terms of the rules above) but are included for convenience.

$$\text{EXPAND_DEF : thm list -> thm -> thm}$$

$$\text{EXPAND_DEF}$$

$$[...]; D = \text{devx}.\text{EQS}\cdot D; D''; ...]$$

"[1]; D1 = D \text{rn}[R]"

"[1]; D1 = devx'.\text{EQS'}; D1""

Where $X'$, $\text{EQS'}$ are got from $X$, $\text{EQS}$ by renaming according to $R$. This is
useful for creating copies of generic devices with different line names.

```
RENAME_LINES : thm -> thm

RENAME_LINES
"|- t == [ | (devX1.EQS1;NXT1)rn[R1]]
   ...
   | (devXm.EQSrn;NXTrn)rn[Rrn] | ] hide L"
"|- t == [ | devX1'.EQS1';(NXT1' rn[R1])]
   ...
   | devXm'.EQSrn',(NXTrn' rn[Rrn]) | ] hide L"
```

Where the primed components are got from the corresponding unprimed ones by renaming.

```
COMBINE_EQUATIONS : thm -> thm

COMBINE_EQUATIONS
"|- t == [ | dev X1.EQ1;N1 | ... | dev Xn.EQn;Nn | ] hide L"
"|- t == dev (X1u ... uXn)\L.EQ1u ... uEQn;([ | N1 | ... | Nn] hide L)"
```

This generates a behaviour equation for an implementation from the behaviour equations for the components (u denotes set union and \ set subtraction).

```
UNWIND_EQUATIONS : tok list -> thm -> thm

UNWIND_EQUATIONS
L
"|- t == devX.EQS;NXT"
"|- t == devX.EQS';NXT'"
```

Where EQS' and D' are got from EQS and D by unwinding on L.

```
PRUNE_EQUATIONS : thm -> thm

PRUNE_EQUATIONS
"|- t == devX.EQS;NXT"
"|- t == devX.EQS';NXT'"
```

Where EQS' is got form EQS by pruning.

The ML functions EXPAND_IMP and VERIFY are defined by:
let EXPAND_IMP L prims imp =
let th1 = UNFOLD_IMP prims imp
in
let th2 = RENAME_LINES th1 ? th1
in
let th3 = COMBINE_EQUATIONS th2
in
let th4 = FOLD imp th3
in
let th5 = UNWIND_EQUATIONS L th4
in
PRUNE_EQUATIONS th5;;

let VERIFY prims spec imp =
UNIQUENESS spec (EXPAND_IMP nil prims imp);;

The next collection of rules enable a certain amount of evaluation of
constants to be done. For each kind of evaluation we provide a simple
rule and a formula conversion (details of conversions are in [Paulson1]).
Only the simple rules are used in the examples described in this report.
The conversions will be useful when we come to define tactics for LSM
(see [Gordon et. al.] and [Paulson] for a description of goal-directed
proof using tactics).

We list below a name and the corresponding evaluation, for example:

\[\begin{align*}
F00 & \quad t_1 \rightarrow t_1' \\
& \quad t_2 \rightarrow t_2'
\end{align*}\]

this means that there are two ML functions:

\[\begin{align*}
F00\_RULE & \quad : \text{thm} \rightarrow \text{thm} \\
F00\_FCONV & \quad : \text{form} \rightarrow \text{thm}
\end{align*}\]

The rule \(F00\_RULE\) will take a theorem \(th\) to \(th'\) and \(F00\_FCONV\) will
take a formula \(fm\) to the theorem \(fm \iff fm'\), where \(th'\) and \(fm'\) are
got from \(th\) and \(fm\) respectively by replacing all subterms of the form \(t1\)
and \(t2\) by the corresponding ones of the form \(t1'\) and \(t2'\): We use \(t, t1, t2,\)
\ldots to range over OL terms.

\[\begin{align*}
BITS & \quad \text{BITS}(\#b_1 \ldots b_w) \quad \rightarrow \{t_1; \ldots; t_w\}
ADD & \quad m+n \quad \rightarrow r
\end{align*}\]

(where \(t_i\) is \(T\) if \(b_i\) is \(1\) and \(F\) otherwise)

\[\begin{align*}
DIF & \quad m-n \quad \rightarrow r
\end{align*}\]

(where \(r\) is the numeral denoting the sum of \(m\) and \(n\))

\[\begin{align*}
EQ & \quad x=x \quad \rightarrow T \\
& \quad x=y \quad \rightarrow F
\end{align*}\]

(where \(x\) and \(y\) are distinct constants)
\[ EL \ i [t_n; \ldots; t_0] \rightarrow t_i \]

\[ WORD \ \text{WORD}_w \ n \rightarrow b_1 \ldots b_w \]

(where \( b_1 \ldots b_w \) is the \( w \)-bit binary representation of \( n \))

\[ VAL \ \text{VAL}_w \ b_1 \ldots b_w \rightarrow n \]

(where \( n \) is the number denoted by \( b_1 \ldots b_w \))

\[ V \ V [t_1; \ldots; t_m] \rightarrow n \]

(where \( n \) is the number denoted by \([t_1; \ldots; t_m]\))

\[ SEQ \ \text{SEQ}(i, j)[t_m; \ldots; t_0] \rightarrow [t_j; \ldots; t_i] \]

\[ AND \ a_1 \ldots a_w \ AND \ b_1 \ldots b_w \rightarrow c_1 \ldots c_w \]

(where \( c_i \) is the value of the conjunction of \( a_i \) and \( b_i \))

\[ OR \ a_1 \ldots a_w \ OR \ b_1 \ldots b_w \rightarrow c_1 \ldots c_w \]

(where \( c_i \) is the value of the disjunction of \( a_i \) and \( b_i \))

\[ NOT \ NOT \ a_1 \ldots a_w \rightarrow b_1 \ldots b_w \]

(where \( b_i \) is the negation of \( a_i \))

\[ COND \ (T \rightarrow t_1 | t_2) \rightarrow t_1 \]
\[ (F \rightarrow t_1 | t_2) \rightarrow t_2 \]

\[ U \ \text{FLOAT}_w \ u \ w t \rightarrow t \]
\[ t \ \text{FLOAT}_w \rightarrow t \]

\[ TRI \ \text{DEST} \ TRI_w (MK \ TRI_w \ t) \rightarrow t \]

\[ \text{bool} \]
\[ t \ \text{AND} \ T \rightarrow t \]
\[ t \ \text{AND} \ F \rightarrow F \]
\[ T \ \text{AND} \ t \rightarrow t \]
\[ F \ \text{AND} \ t \rightarrow F \]
\[ t \ \text{OR} \ T \rightarrow T \]
\[ t \ \text{OR} \ F \rightarrow t \]
\[ T \ \text{OR} \ t \rightarrow T \]
\[ F \ \text{OR} \ t \rightarrow F \]
\[ NOT \ T \rightarrow F \]
\[ NOT \ F \rightarrow T \]

For example, \( ADD\_RULE \) would reduce \( \lceil t = (2+3)+4 \rceil \) to \( t = 9 \), and \( bool\_FCONV \) would map the formula \( NOT \ T = x \ OR \ F \) to the theorem:

\[ NOT \ T = x \ OR \ F \iff F = x \]

49
References

[Ayres]
R. Ayres. *VLSI: silicon compilation and the art of automatic microchip design.* Prentice-Hall, 1983

[Barrow]
H. Barrow. *Proving the Correctness of Digital Hardware Designs.*
Available from: Dr. Harry Barrow, Fairchild Laboratory for Artificial Intelligence Research, 4001 Miranda Ave., Palo Alto, CA 94304.

[Dembinski & Budkowski]

[Gordon et. al.]

[Gordon1]
M. Gordon. *The Denotational Semantics of Sequential Machines.*
Information Processing Letters, Volume 10, Number 1, 1980

[Gordon2]

[Gordon3]
M. Gordon. *A Very Simple Model of Sequential Behaviour of nMOS.*

[Gordon4]

[Gordon5]

[Hanna]

[Milne]
Acknowledgements

I have had useful conversations with Harry Barrow, John Herbert, George Milne, Robert Milne, Ben Moszkowski, Larry Paulson and Edmund Ronald. Tom Melham made valuable comments on a first draft of this paper.