A Language to Support Verification of Embedded Software

Riaan Swart

THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH.

Supervised by: Prof P.J.A. de Villiers

April 2004

Stellenbosch University http://scholar.sun.ac.za

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Stellenbosch University http://scholar.sun.ac.za

Abstract

Embedded computer systems form part of larger systems such as aircraft or chemical processing facilities. Although testing and debugging of such systems are difficult, reliability is often essential. Development of embedded software can be simplified by an environment that limits opportunities for making errors and provides facilities for detection of errors. We implemented a language and compiler that can serve as basis for such an experimental environment. Both are designed to make verification of implementations feasible.

Correctness and safety were given highest priority, but without sacrificing efficiency wherever possible. The language is concurrent and includes measures for protecting the address spaces of concurrently running processes. This eliminates the need for expensive run-time memory protection and will benefit resource-strapped embedded systems. The target hardware is assumed to provide no special support for concurrency. The language is designed to be small, simple and intuitive, and to promote compile-time detection of errors. Facilities for abstraction, such as modules and abstract data types support implementation and testing of bigger systems.

We have opted for model checking as verification technique, so our implementation language is similar in design to a modelling language for a widely used model checker. Because of this, the implementation code can be used as input for a model checker. However, since the compiler can still contain errors, there might be discrepancies between the implementation code written in our language and the executable code produced by the compiler. Therefore we are attempting to make verification of executable code feasible. To achieve this, our compiler generates code in a special format, comprising a transition system of uninterruptible actions. The actions limit the scheduling points present in processes and reduce the different interleavings of process code possible in a concurrent system. Requirements that conventional hardware places on this form of code are discussed, as well as how the format influences efficiency and responsiveness.

Stellenbosch University http://scholar.sun.ac.za

Opsomming

Ingebedde rekenaarstelsels maak deel uit van groter stelsels soos vliegtuie of chemiese prosesseerfasiliteite. Hoewel toetsing en ontfouting van sulke stelsels moeilik is, is betroubaarheid dikwels onontbeerlik. Ontwikkeling van ingebedde sagteware kan makliker gemaak word met 'n ontwikkelingsomgewing wat geleenthede vir foutmaak beperk en fasiliteite vir foutbespeuring verskaf. Ons het 'n programmeertaal en vertaler geïmplementeer wat as basis kan dien vir so 'n eksperimentele omgewing. Beide is ontwerp om verifikasie van implementasies haalbaar te maak.

Korrektheid en veiligheid het die hoogste prioriteit geniet, maar sonder om effektiwiteit prys te gee, waar moontlik. Die taal is gelyklopend en bevat maatreëls om die adresruimtes van gelyklopende prosesse te beskerm. Dit maak duur looptyd-geheuebeskerming onnodig, tot voordeel van ingebedde stelsels met 'n tekort aan hulpbronne. Daar word aangeneem dat die teikenhardeware geen spesiale ondersteuning vir gelyklopendheid bevat nie. Die programmeertaal is ontwerp om klein, eenvoudig en intuïtief te wees, en om vertaaltyd-opsporing van foute te bevorder. Fasiliteite vir abstraksie, byvoorbeeld modules en abstrakte datatipes, ondersteun implementering en toetsing van groter stelsels.

Ons het modeltoetsing as verifikasietegniek gekies, dus is die ontwerp van ons programmeertaal soortgelyk aan dié van 'n modelleertaal vir 'n modeltoetser wat algemeen gebruik word. As gevolg hiervan kan die implementasiekode as toevoer vir 'n modeltoetser gebruik word. Omdat die vertaler egter steeds foute kan bevat, mag daar teenstrydighede bestaan tussen die implementasie geskryf in ons implementasietaal, en die uitvoerbare masjienkode wat deur die vertaler gelewer word. Daarom poog ons om verifikasie van die uitvoerbare masjienkode haalbaar te maak. Om hierdie doelwit te bereik, is ons vertaler ontwerp om 'n spesiale formaat masjienkode te genereer bestaande uit 'n oorgangstelsel wat ononderbreekbare (atomiese) aksies bevat. Die aksies beperk die skeduleerpunte in prosesse en verminder sodoende die aantal interpaginasies van proseskode wat moontlik is in 'n gelyklopende stelsel. Die vereistes wat konvensionele hardeware aan dié spesifieke formaat kode stel, word bespreek, asook hoe die

Stellenbosch University http://scholar.sun.ac.za

formaat effektiwiteit en reageerbaarheid van die stelsel beïnvloed.

Stellenbosch University http://scholar.sun.ac.za

Acknowledgements

I gladly acknowledge the help of several people who made this project feasible:

- Pieter de Villiers, for guidance throughout my University career and especially the last two years.
- Leon Grobler, who is currently implementing the runtime system, for continued assistance.
- My parents, for supporting and encouraging me.
- The guys in the Hybrid Laboratory, for the great working environment and much comic relief.

Stellenbosch University http://scholar.sun.ac.za

Contents

Abstract iii

Opsomming iv

Acknowledgements vi

1 Introduction 1

1.1 Outline of the thesis . . . 3

2 Literature Survey 4

2.1 CSP-based computer languages . . . 5

2.2 occam . . . 5

2.2.1 Structure of an occam program . . . 6

2.2.2 Primitive processes . . . 6

2.2.3 Sequential and parallel execution and control flow . . . 7

2.2.4 A construct for nondeterminism . . . 7

2.2.5 Named processes . . . 8

2.2.6 Sharing of variables and channels . . . 9

2.2.7 Evaluation of language . . . 9

2.3 Joyce . . . 10

Stellenbosch University http://scholar.sun.ac.za

2.3.1 Processes in Joyce 10
2.3.2 Channels and interprocess communication 11
2.3.3 Non-determinism in Joyce 12
2.3.4 Evaluation of language 14
2.4 occam 2 15
2.4.1 Types and type coercion 15
2.4.2 Channel protocols 16
2.4.3 Other features 16
2.4.4 Evaluation of language 17
2.5 occam 3 17
2.5.1 Modules 18
2.5.2 Libraries 19
2.5.3 Evaluation of language 19
2.6 Model checking 20

3 The LF Language 21

3.1 Previous work 21
3.2 Design goals for the new version 23
3.2.1 Base the language on CSP 23
3.2.2 Eliminate language features to make model checking feasible 23
3.2.3 Safe programming practices 23
3.2.4 Intuitive and easy to understand language 24
3.2.5 Small runtime system 24
3.2.6 Low-level operations 24
3.2.7 Context switching and interprocess communication in software 24

Stellenbosch University http://scholar.sun.ac.za

3.3 Processes 25

3.3.1 Design considerations 25

3.4 Types, variables and constants 25

3.4.1 Design considerations 27

3.5 Expressions, assignments and control structures 27

3.6 Process instantiation 29

3.6.1 Design considerations 29

3.7 Interprocess communication 32

3.7.1 Channels and ports 32

3.7.2 Sending and receiving messages 33

3.7.3 Design considerations 35

3.8 A construct for nondeterminism 39

3.8.1 Design considerations 40

3.9 Modules 43

3.9.1 Design considerations 44

3.10 Constructs to facilitate low-level programming 45

3.10.1 Communicating with peripherals via memory 45

3.10.2 Time measurement in LF 46

3.11 The SYSTEM module 47

3.11.1 Interrupts 47

3.12 Scoping and concurrent access 48

3.13 Security claims 49

3.14 Some LF examples 51

3.14.1 Generate a bounded stream 51

Stellenbosch University http://scholar.sun.ac.za

3.14.2 Copy a bounded stream 51
3.14.3 Merge a bounded stream 52
3.14.4 Suppress duplicates in a bounded stream 52
3.14.5 Library 53
3.14.6 Abstract data structure 54
3.14.7 Input/output 55
3.15 Modifications to the language for this thesis 57
3.15.1 Limiting user freedom to promote safety 57
3.15.2 Improved control over concurrency 57
3.15.3 Abstraction 57
3.15.4 Constructs to support hardware-level programming 58
3.16 Summary 58

4 Code Generation to Support Model Checking 59

4.1 Goals 60
4.1.1 The Intel 386 processor and assembly language 60
4.2 Transitions generated by the LF compiler 61
4.2.1 The interpreter 62
4.3 Implementation of the compiler 64
4.3.1 Symbol table 64
4.3.2 Tree structure and code generation 69
4.3.3 Process activation record layout 81
4.3.4 Process activation 83
4.3.5 Modules and separate compilation 84
4.4 Regression testing 86

Stellenbosch University http://scholar.sun.ac.za

4.5 Overhead of the interpreter 88
4.6 Summary 91

5 Conclusion **92**

5.1 Revision of design goals for the language 92
5.2 The influence of the transition system approach 95
5.3 Future work 95
5.3.1 To improve the efficiency of the transition system 95
5.3.2 Other improvements 98
5.4 Final thoughts 98

A EBNF of LF **99**

B Intrinsic processes **102**

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

Introduction

Computer systems have spread from the desktop to almost every aspect of our lives. The demands on these technologies are always increasing as the applications of computers broaden. As companies design more complex software for increasingly varying purposes, good software design methodologies and thorough testing are often not adequate to ensure software reliability any more.

These statements are particularly relevant for the development of embedded systems. A computer system can be described as 'embedded' if it is integrated into a bigger (non-computer) system; examples are the ABS brakes on modern motor vehicles and control systems in aircraft. Output facilities for embedded systems are often limited. Testing of such systems can be difficult and, with concurrent systems, not adequate to detect subtle concurrency errors. Yet reliability is important for such applications and when failure does occur, correcting the error can be expensive or impossible.

An environment which could minimise the opportunities for making errors, and make it easier to detect some errors early, would simplify software development. The aim of this thesis is to describe the language and compiler for such a development environment. The work described here forms part of a bigger project, the aim of which is to design a complete system for faster development of reliable embedded systems.

Efficiency is always important in resource-strapped embedded systems, but not more so than correctness and safety. Therefore, our first design priority was correctness and safety, although efficiency was also a consideration. The language LF was thus designed to encourage correctness and detect errors. Run-time safety checks such as range checking for array indexing were viewed as important to include where needed. However, such checks only *indicate*

Stellenbosch University http://scholar.sun.ac.za

run-time errors (mostly by aborting execution); they do not prevent them. The language should prevent as many coding errors as possible by discouraging unsafe coding practices, and create opportunities for the compiler to detect coding errors wherever possible.

Such language measures are ways to prevent some coding errors from occurring in software. Yet most logical errors, especially for concurrent software, cannot be detected in this way. A technique which promises to be a practical approach to verifying correctness properties for concurrent software is model checking. A model is created of the system being verified, abstracting away detail, yet retaining the control flow to mechanically verify correctness properties. Every value that every variable in the system can assume is determined and kept record of, forming the *state space* of the system. Correctness claims are specified, for example that an implementation of a communication protocol will always acknowledge a received packet. The model checker then traverses the state space to find states where such claims are false.

This technique has been successful in finding errors in concurrent software, but errors can still occur. For example, mistakes might be made in the derivation of a model from the implementation source code. Compiler errors might also introduce errant behaviour in the executable code. Therefore, even if correctness properties were verified for a model, they might not hold for the implementation. To eliminate these sources of errors, model checking has to be done at the compiled machine code level.

To study these issues, this project aims:

- to design a language called LF, and implement a compiler for that language, to implement less error-prone embedded systems.
- The language should simplify the detection of errors through means such as model checking or run-time verification. By making detection of errors easier, the environment should allow faster development of quality embedded software. One of the languages which influenced the design of LF is Promela, the modelling language of the model checker Spin. Spin is a widely used model checker, and Promela is close to a programming language. A run-time system needed for a language similar to Promela is small and relatively simple to implement, and is therefore well-suited for embedded systems.
- Executable code is generated in a form to make it possible to generate states for a model checker from the executables. Some execution overhead is involved, but ways are suggested in which the efficiency of such executables could be improved.

Stellenbosch University http://scholar.sun.ac.za

- Since LF is intended for implementing embedded systems, specialised constructs are needed for programming at the hardware level.

The work in this thesis is based on work previously done by Van Riet [21]. He designed the first prototype version of LF for embedded work, based on the language Joyce [8]. He also implemented a runtime system to support the language. The project described here focuses on the language and the form of the executable code (a new runtime system and a model checker for the LF system are the subjects of separate studies currently underway). The language has been modified and enhanced. For example, LF code is now written in separately compilable modules, and the interprocess communication constructs have been generalised. The prototype compiler described in [21] has been discarded and replaced in this project with a two-pass compiler which can form the basis for future work. For example, since overhead is involved in executing the LF code, optimisations on the intermediate code format can be implemented to minimise this overhead.

1.1 Outline of the thesis

Chapter 2 is an overview of a number of languages that influenced the design of LF. Several concurrent languages are briefly discussed, including several implementation languages and one language for model checking. Then, two implementation languages for concurrent software are examined in some detail and compared with one another in terms of factors such as design goals, safety and efficiency.

Chapter 3 describes the LF language. A brief overview of the original prototype of the language is given, and the design goals of the new version of the language are set out. Then the language is discussed; examples of all the constructs are given, and design decisions are discussed. Claims about the security provided by the language are discussed.

Chapter 4 discusses the implementation of the LF compiler and how the code is generated to simplify model checking. The format of the code generated by the LF compiler, and how this code is executed, is described. The design of the compiler is outlined, and the focus falls on what machine code is generated for every LF construct. Then the overhead involved in executing the special format of code is discussed.

Finally, Chapter 5 summarises, reviews and evaluates the work done. The language is reviewed as an implementation language and as a language to support model checking. The influence of the special form of executable code is evaluated, and future work is outlined.

Stellenbosch University http://scholar.sun.ac.za

Chapter 2

Literature Survey

A typical embedded system needs to interact with multiple external devices. Concurrency is an efficient and elegant way to implement such functionality. However, concurrent designs are often fraught with subtle errors due to the complex interaction between different components. This is illustrated, for example, by the well publicised Therac-25 accidents where concurrency errors led to the death of patients [15, Appendix A]. Much effort has therefore been devoted to the problem of developing reliable concurrent software.

One framework for concurrent programming that has had a big impact is called *CSP*, or *Communicating Sequential Processes* [10]. In this framework, communication between concurrently executing processes is based on synchronous message passing. No data can be shared between processes; synchronous message passing is therefore the *only* way in which a process can influence the control flow of other processes.

Because data cannot be shared among processes, the order in which processes (and operations on variables) are scheduled cannot cause corruption of data. Processes will not wait for access to shared data as when shared data is protected by constructs such as monitors or semaphores. However, message passing overhead will be more in a CSP-based system than in a system where shared data is protected by monitors or semaphores. A system of concurrent communicating processes has to be designed with these considerations in mind.

Stellenbosch University http://scholar.sun.ac.za

2.1 CSP-based computer languages

This chapter gives an overview of languages based on CSP, and then focuses on the two languages that most influenced the design of LF, the language introduced in this thesis.

CSP inspired several new implementation language designs after it was first described in 1978, but most languages did not implement all the features and qualities of the specification system.

RBCSP [18] does not allow shared data, but implements buffered communication in contrast with CSP. Even individual commands can be specified to execute concurrently.

The low-level language occam [13, 17] adheres to the CSP principles of processes communicating *only* via synchronous message passing. Write access to data shared between concurrent processes is not allowed. The initial version discussed in [17] is typeless, but a later version introduces types [14]. The language is intended for programming embedded systems [14, Chapter 7].

Planet is intended for distributed systems [7], an environment for which the message passing paradigm is particularly suited. Some sharing of memory is allowed in Planet — process definitions can be nested, and every process shares its variables with the processes nested within it. The syntax of Planet is based on Pascal.

Joyce [8] is another Pascal-like language for the design and implementation of distributed systems. It implements no shared data and synchronous communication, like CSP.

Promela is not an implementation language, but the modelling language for the model checker Spin [12]. As a modelling language it must be able to represent behaviour of systems implemented in other implementation languages. Interprocess communication in Promela is inspired by the CSP model, yet global variables are allowed and buffered communication is supported.

2.2 occam

The language occam was designed as the native language for the INMOS transputer. Hardware support for interprocess communication convinced the designers that occam would lead to an "unaccustomed programming style" [13, Preface], where massive networks of communicating processes would perform many tasks in parallel.

Stellenbosch University http://scholar.sun.ac.za

Experience in using occam to build systems led to two new versions of the language. This section focuses on the first version, or 'proto-occam' [14, Chapter 1]; subsequent sections will discuss the later releases. The discussion serves to give a general idea of the language; for a detailed discussion of occam refer to [13].

2.2.1 Structure of an occam program

An occam program is written as a sequence of lines; there is no symbol that ends or separates commands, except for the end-of-line character. Indentation is used in occam to indicate nesting of commands.

2.2.2 Primitive processes

In occam *all* commands are viewed as processes. The simplest of these are two processes that do nothing, `SKIP` and `STOP`. `SKIP` is always executable, and does nothing except terminate. `STOP` also does nothing, but it never terminates. It is used to bring execution of a composite sequential process to a halt without affecting other independent processes.

Assignments in occam have the same form as in conventional, Algol-like languages, namely:

```
variable := expression
```

Channels are used to connect two occam processes for communication. This differs from the 1978 version of CSP on which the language is based, and where the communication partner must be named directly. Communication in occam is implemented by the *send* and *receive* primitive processes, as illustrated by the following example. Comments at the end of each line, started with the symbol "--", indicate the function of each line.

```
CHAN fromProducer, toConsumer: -- declare channel
-- ...
fromProducer ? result -- receive result
-- ...
toConsumer ! result -- send result
```

In the declaration of a channel, the `CHAN` keyword is followed by a list of the channels being declared. When receiving a message, the name of the channel is followed by a "?", and the

Stellenbosch University http://scholar.sun.ac.za

variable into which the message will be copied. Similarly, when sending a message, the name of the channel is followed by a "`!`", and the value that is to be sent.

The occam language supports only synchronous communication. A channel can also transmit messages in only one direction and between only two processes. For two-way communication two channels are needed, and a client-server architecture needs a separate channel from the server to every client. Mechanisms such as broadcasting need to be implemented by the programmer.

2.2.3 Sequential and parallel execution and control flow

Individual commands can be executed in sequence or in parallel. The `SEQ` and `PAR` keyword respectively specify sequential and parallel execution of commands.

Examples:

```
0   SEQ -- execute indented commands in sequence
1     x := a + b
2     y := y + 3
3   -- ...
4   PAR -- execute indented commands in parallel
5     a := b + c + 1
6     WriteToScreen(x)
```

The commands below the `SEQ` keyword (lines 1–2) are executed in sequence, and all commands below the `PAR` keyword (lines 5–6) are executed in parallel.

The `IF` and `WHILE` in occam function like their counterparts in conventional, Pascal-like languages.

2.2.4 A construct for nondeterminism

In CSP, the "$\Box$" operator introduces nondeterminism into the control flow. This is useful where a process is waiting to react to one of several possible events, such as a server awaiting requests from different clients, each associated with an event or *guard*. The process whose guard becomes executable (event happens) first, is executed. If c and d were events and P and Q were processes, this would be written as $c \rightarrow P \Box d \rightarrow Q$.

Stellenbosch University http://scholar.sun.ac.za

occam defines the ALT (alternative) construct as an analogue to the CSP $\square$ operator. An example of the occam ALT construct is given below. Contrary to the CSP operator, a guard in an ALT process cannot be any action; it must contain either SKIP or an input process, and it may also include an expression — nothing else. A guard consisting of only SKIP is always enabled. A guard consisting of only an input process is enabled if the input is possible. If the guard includes an expression, the guard is enabled if the expression evaluates to TRUE and the rest of the guard is a SKIP or an enabled input, for example:

```
0  ALT
1    booleanCondition1 & ch1 ? x
2      out ! x
3    booleanCondition2 & ch2 ? x
4      out ! x
```

The guards are on lines 1, and 3. If a guard is enabled, the indented lines below it are executed (lines 2 or 4). This command accepts a value from channel ch1 if booleanCondition1 is TRUE, or from channel ch2 if booleanCondition2 is TRUE. In both cases, the received value will be output on channel out.

2.2.5 Named processes

Any occam process can be given a name. A named process can be instantiated by writing the name of the process, like a procedure call in a sequential language. However, a named process can only be instantiated once its declaration is finished, so recursion is impossible in occam.

An example:

```
PROC Add(VALUE arg1, arg2) =
VAR answer :
SEQ
  answer = arg1 + arg2
  PrintScreen(answer)
```

This example defines a named process Add, with two parameters called arg1 and arg2. The two parameters are added and displayed on the screen. Note that a colon follows after the declaration of variables and constants, as the VAR answer : in the example shows.

Stellenbosch University http://scholar.sun.ac.za

2.2.6 Sharing of variables and channels

Usage rules are defined in occam to ensure that race conditions are avoided. For example, all components of a PAR construct are allowed to read a shared variable as long as no component writes to the variable. If a component does write to a shared variable, only that component may access the variable for reading or writing.

Another condition that must always be true in occam is that a channel may connect only one sender to one receiver. An example of a usage rule to enforce this is that only one component of a PAR command may output over a specific channel and only one other component of the same PAR may input over the same channel.

Parameters passed to named processes amount to either sharing or copying of data, and is also subject to usage rules, depending on the kind of parameter:

- A VALUE formal parameter is viewed as a run-time constant in the procedure body.
- A VAR formal parameter renames the actual parameter in the process body, like a reference parameter in Pascal. Therefore the named process and its caller shares the variable.
- A CHAN formal parameter denotes the passing of a channel as parameter, so the named process and its caller share the channel.

Since a variable may not be concurrently changed by two or more processes, and a channel may only connect two concurrently running processes, VAR and CHAN parameters implies sharing of variables and channels respectively, so named processes with such parameters will be restricted in the way in which they can be instantiated. Also, no process may change the value of a variable passed as VALUE parameter while the named process that received it can still be executing.

2.2.7 Evaluation of language

Intricate tasks can be subdivided into simpler concurrent tasks in occam, so elegant solutions to problems such as matrix multiplication can be expressed. Message passing in occam also simplifies tasks such as interrupt handling. The designers of occam claim that message passing between occam processes are "right for a task for which interrupt handling routines have always been inadequate" [13, Chapter 5]. The ALT construct also proves useful for specifying a choice of operations depending on the environment of a process (see section 2.2.4).

Stellenbosch University http://scholar.sun.ac.za

The novel design of occam was inspired by inexpensive communication and context switching operations on the INMOS transputer. However, such operations *are* expensive on most conventional architectures.

Another criticism of the language was that it is so low-level that it is only practical for small or critical applications [20]. The inability to pass complex objects (arrays or records) in a single message in occam was also criticised [2].

One noteworthy aspect of occam is the language rules introduced to avoid and detect programming mistakes, especially concurrent errors, at compile time. Examples of such rules were given in Section 2.2.6.

2.3 Joyce

Also inspired by CSP, Joyce is described as a "secure programming language" for "the design and implementation of distributed systems" by its creator Brinch Hansen [8]. The syntax, based on Pascal, is more conventional than that of occam. The language also differs from occam in that complete type checking of variables and messages is done during compilation, and processes can be instantiated recursively.

Joyce is simple and elegant in design and more suited to conventional hardware architectures than occam. However, where occam was used for several big projects, Joyce did not have a similar impact. Nevertheless, a number of important ideas were included in the language. Some of these concepts influenced the design of LF, and therefore a brief discussion of Joyce is considered relevant.

2.3.1 Processes in Joyce

An example that shows process instantiation, creation of channels and interprocess communication is shown below. Processes in Joyce are called *agents.*

```
agent Factorial(m, n: integer);
begin
  if n = 1 then Output(m)
  else Factorial(m*n, n - 1);
end;
```

Stellenbosch University http://scholar.sun.ac.za

This agent calculates the factorial of its parameter n, and outputs it; the parameter m should initially be 1. The answer is output to the screen and not returned to the caller because Joyce does not support reference parameters, and does not implement function processes. Reference parameters would allow a process to reference data of its caller, and shared data is not allowed in Joyce. Function processes are not implemented, because all processes execute concurrently, and there is no way to know when a process would terminate and return its result.

2.3.2 Channels and interprocess communication

Similar to occam, Joyce agents send messages via channels without naming the recipient directly. However, Joyce channels, like agents, are allocated dynamically. Joyce also differs from occam in that messages are typed, and several different kinds of messages can be sent over a single channel. Each kind of message is denoted by a symbol. A process ready to receive a message with a given symbol can only communicate with a sender that sends a message with the same symbol. The examples below show how channel types are declared:

```
type
  Alphabet1 = [Symbol1(integer),
    Symbol2(char)];
  Alphabet2 = [Signal];
```

Each symbol can contain either a single typed value or no data at all. For example, the `Alphabet1` type has symbols `Symbol1` containing an integer value and `Symbol2` containing a `char` value. The `Alphabet2` type has only one symbol, `Signal`, containing no data. The collection of all symbols that can be sent over a channel is called the *alphabet* of the channel.

Agents declare *port variables* to store references to channels. During declaration of the port variables, a channel type is associated with each port declared, as illustrated below:

```
var
  port : Alphabet1;
```

An agent allocates a channel during run-time by executing a command as shown below. As a result, a reference to the new channel is stored in the port variable.

```
  +port;
```

Stellenbosch University http://scholar.sun.ac.za

A process that sends a symbol over a channel is blocked until a process receives the same symbol over the same channel. This scheme contributes to the security of communication over the channel. A message sent by one process cannot be misunderstood by the receiver; only messages of the right type will be received.

Examples of sending and receiving in Joyce:

```
port ? Symbol1(x);
port ! Symbol2(ch)
```

Other differences between occam channels and Joyce channels are that Joyce channels can be shared, and that Joyce allows an agent to send and receive on the same channel. However, communication is still synchronous and between only one sender and receiver. Therefore only one receiver can receive a message from only one sender, and the transfer only takes place when both agents are ready to communicate. Joyce limits neither the number of agents communicating over a certain channel, nor the direction in which an agent communicates over a channel. If two agents send the same symbol on the same channel and only one agent receives it on that channel, the receiver will be non-deterministically matched up with one of the senders.

2.3.3 Non-determinism in Joyce

Joyce agents are selected for execution according to a scheduling strategy which is not known at compile time. The scheduler will influence which processes are ready to send and receive in an unpredictable way. Channels can be shared between more than two processes in Joyce, so the scheduler will influence which messages are received by which processes, and the order of reception. Control flow in Joyce programs are therefore subjected to the influences of the scheduler in ways not possible in occam.

Further nondeterminism is also present in Joyce channels, as described in section 2.3.2. Suppose two processes are waiting to send the same symbol over the same channel. When a receiver receives this symbol over the channel, only one sender will be arbitrarily selected to communicate with the receiver.

Stellenbosch University http://scholar.sun.ac.za

A construct for explicit nondeterminism

As most CSP-based languages, Joyce implements an analogue to the CSP "□" operator — the *polling* command. The ALT construct in occam has a similar function. The polling command blocks until a guard is executable, and if a guard is executable the command list following the guard is executed. A guard can only consist of a communication command with an optional boolean expression. For the guard to be executable the communication must be possible, and if the boolean expression is present, it should be True. The guards will be tested cyclically until an executable guard is found.

The example below shows a Joyce agent containing a simple poll command:

```
0  agent merge(in1, in2, out : stream);
1  var x : integer;
2  begin
3    while true do
4      poll
5        in1 ? int(x) -> out ! int(x)
6        in2 ? int(x) -> out ! int(x)
7      end
8    end
9  end;
```

Agent merge accepts input from two different channels, and outputs it on a single channel.

To illustrate the differences between the occam ALT and the Joyce poll, the example above is extended. In the example below, the agent only accepts input values if the value is less than 10. The agent also maintains a count of values accepted. If the count is requested over the output channel, the agent will output it and reset the count to 0.

```
0  agent mergeAndCount(in1, in2, out : stream);
1  var x, count : integer;
2  begin
3    count := 0;
4    while true do
5      poll
6        in1 ? int(x) & x < 10 ->
```

Stellenbosch University http://scholar.sun.ac.za


```
7           count := count + 1;  out ! int(x)
8         in2 ? int(x) & x < 10 ->
9           count := count + 1;  out ! int(x)
10        out ! int(count) -> count := 0
11      end
12  end;
```

The Joyce `poll` differs from the occam `ALT` in the following ways:

- The boolean expression in the first two guards (line 6 and 8), known as a *conditional receive*, contains variables that will be changed by reception of the message. In these boolean expressions, the value of x will be the value *to be received*, not the current value. This allows an agent to examine a message before receiving it.
- The third guard (line 10) of the `poll` command contains an output command. The occam `ALT` construct may not have such an output command as part of a guard, to prevent `ALT` guards of different processes on the same channel from matching. For the same purpose, Joyce send and receive guards within a `poll` can only match with send and receive commands which are not `poll` guards.

2.3.4 Evaluation of language

Because Joyce was never widely used, no evaluation of the language based on usage experience is available. However, several observations can be made.

Channel sharing, symbols (section 2.3.2), and conditional receive when polling (section 2.3.3) simplify interfaces between processes. Communication is made more secure by typed messages. Simplicity and security are factors that assist the programmer when implementing projects of any size. They are therefore desirable in any implementation language. Some features of Joyce introduce overhead and inefficiency into the language, for example dynamic agent instantiation. However, because Joyce is targeted at distributed systems and not embedded systems, efficiency is less of a concern than maintainability and scalability. As more powerful hardware becomes available, the advantages offered by these features should eventually outweigh the disadvantages.

One aspect of Joyce causes overhead that is deemed unnecessary. This is that every Joyce agent is allowed to access only its own variables. All agents in Joyce execute in parallel, and

Stellenbosch University http://scholar.sun.ac.za

Joyce allows no sharing of data between agents. Therefore, much copying of data (through message passing) is needed in any non-trivial application. For example, if a programmer wants to define a subroutine for a task that is often performed in an agent, he or she will need to define another agent. All data that the 'subroutine' agent operates on will have to be copied to the 'subroutine' and back because there are no reference parameters or shared data in Joyce. If programmers want to avoid this overhead, they have to make do without subroutines, an important abstraction tool.

In contrast, occam makes it possible to specify parallel and sequential execution of commands. Intricate usage rules allow processes to share data *only* if they do not execute in parallel. Sequential programmers are not used to such rules, but these allow the compiler writer to ensure safe concurrent access to data.

2.4 occam 2

An important deficiency of the original version of occam discussed in section 2.2 is lack of types and type checking for variables and messages. Brinch Hansen commented in [8] that "In this respect, CSP and occam are insecure languages." Typing was thus the most important addition to occam in version 2 (discussed in [14]) and version 2.1 (described in [16]), that emerged after Joyce. In this section we overview the additions in occam 2 and occam 2.1 most relevant to our study.

2.4.1 Types and type coercion

In all occam versions since version 2, variables and constants must have types. Basic (built-in) data types include booleans, bytes, signed integers stored in 8, 16 and 32 bits, as well as IEEE single and double precision floating point types. Structured data types can be arrays (not restricted to one dimension as in occam 1), and as of occam 2.1, also records.

All the operands of an operator must exactly match the type for which the operator is defined; no implicit type conversion is done in occam. *Type coercion* must be used to convert an expression of one type to another type.

Stellenbosch University http://scholar.sun.ac.za

2.4.2 Channel protocols

A computer 'protocol' is usually a specification of how different computers can communicate; in occam, protocols specify how different processes communicate. This is an example of how occam encourages programmers to think about processes as running concurrently, each on its own processor and with its own memory. Just as variables and constants are typed, channels must all have types called *protocols* associated with them during declaration. If a channel has a certain protocol, all the values sent over that channel must match the types described in the protocol.

Simple and sequential protocols

The simplest protocols are just types of variables. If a channel is associated with a simple protocol, only a single value of a certain type can be sent over the channel at a time. A single value can have a basic type, or a composite (array or a record) type.

Sequential protocols are a composition of several simple protocols. Therefore every message must consist of a series of values in the sequence specified by the protocol.

Discriminated protocols

A channel with a discriminated protocol in occam transmits messages in a similar fashion as channels in Joyce do. Messages in Joyce are associated with symbols; likewise, messages are identified with *tags* in an occam discriminated protocol. In Joyce, communication does not take place if the symbol on the input side and the symbol on the output side do not match. In contrast, communication does take place when this happens in occam. However, the process inputting the message will then behave like the `STOP` process, indicating a run-time error.

2.4.3 Other features

The temporary renaming or *abbreviation* of a constant, variable or part thereof is allowed in occam. Abbreviations can be used, for example, to give names to disjoint segments of an array, and so subdivide it. Each of these disjoint segments of the original array can then be referenced (and updated) by different concurrently running processes. Since the compiler has bounds for each segment, index checking within each segment can ensure that data is not shared between concurrent processes.

Stellenbosch University http://scholar.sun.ac.za

If a *variable* is abbreviated, a new name is associated with that variable throughout the scope of the abbreviation (the part of the program just below the abbreviation, and indented one level). If a *value* is abbreviated, a name is given to an expression and all variables used in that expression must stay constant throughout the scope of the abbreviation.

2.4.4 Evaluation of language

Much functionality has been added to the language, and type checking adds security that has been lacking. It is apparent that efficiency is still a major design consideration. For example, although the discriminated protocol in occam supports typing of messages similar to channel types in Joyce, the run-time system transmits the message regardless of whether or not the tag of the output matches a tag of the input. In contrast, the run-time system of Joyce must decide whether to transmit a message or not, based on the matching of alphabet symbols.

Yet, a conditional receive such as implemented in the Joyce `poll` would be purposeless in occam because occam channels are shared between only two processes. It can be said that the occam programmer has to manually 'indicate' which messages are intended for which processes; this is done by supplying a channel for each type of message that can be sent. Again, the design of occam sacrifices flexibility for efficiency. The occam user is also expected to implement the functionality by creating (sometimes intricate and error-prone) networks of channels. In contrast, the Joyce run-time system implements conditional reception. Either strategy could be beneficial; it depends on the nature of the project to be implemented.

2.5 occam 3

A deficiency of the original occam that was not addressed by occam 2 or occam 2.1 was that it was a low-level implementation language, not fit for the implementation of bigger and more complex systems (Section 2.2.7). To address this deficiency, several tools for abstraction and program structuring is provided in occam 3. The most significant of these are modules and libraries, and a mechanism for remote procedure calls. Again, features relevant to our study are highlighted below.

Stellenbosch University http://scholar.sun.ac.za

2.5.1 Modules

A module in occam is an entity that groups processes together and prevents the rest of the system from addressing or accessing these processes. The module definition is specified with an interface for the module to interact with the rest of the system. As many instances as needed of the module can then be instantiated. Several instances of the same module definition may exist at the same time in a system, and references to module instances can be passed as parameters to procedures (named processes).

Since several instances of a module definition can be in existence at a time, rules are needed to prevent shared variables or channels. For example, a module may only change variables that are defined within the body of the module. The rules for occam 3 modules are discussed in [3, Chapter 13].

Interface types

The definition of a module is viewed as its type — three instantiations of the module definition $M1$ will have the same type $M1$. An instantiation of a different definition $M2$ will have a different type $M2$ from the three $M1$ modules, even though it might have the same interface.

However, interface types in occam allow the declaration of an interface which will serve as the type of the module. Any module which has the same interface as declared in the interface type can be instantiated with the new interface type instead of its own declaration type. This allows modules with different declarations to have a similar type, and can be used to implement polymorphism of modules.

For example, suppose two modules have different declarations but the same interface, and an interface type is defined to match that interface. Suppose also that a process accepts two parameters of the interface type. References to the two different modules can then be passed as parameters of the same interface type to the process. Even though their types match inside the process, the two modules might behave differently.

Modules together with interface types implement much of the functionality of *active objects* in object-oriented languages. Active objects contain data, methods and a thread of execution, whereas occam modules contain data and processes.

Stellenbosch University http://scholar.sun.ac.za

2.5.2 Libraries

Like modules, libraries are intended to provide structure to occam code. Where several instantiations of the same module definition can be in existence at the same time, only one instance of a library is in existence in occam code. Libraries can define private or exported data, procedures (named process definitions) and functions. It is used to implement abstract data types or system services.

Separate compilation and linking

A separate compilation unit in occam 3 is a self-contained library, that uses no entities defined outside its scope. If it is imported into other code, it is instantiated once; if there is internal data in the library, there will be one copy of it in the importing code. The operating system associates the exported entities of a library with the name of the text file in which the library is defined.

2.5.3 Evaluation of language

The purpose of the new constructs in occam 3 was to support more complex implementations ('medium and large programs') [3, Introduction]. To limit complexity in bigger implementations, abstraction and structuring are essential; occam 3 provides much of the functionality provided for this in modern sequential languages. However, the designers have not lost sight of many of their design principles. For example, no dynamic allocation of memory, or dynamic instantiation of processes is implemented. Channels still connect only two processes at a time. The static nature of the language is preserved.

However, implementations in occam need to be edited and recompiled to create new processes and channels. A feature such as dynamic process instantiation will be useful, as it will enable a system, for example, to create more processes and channels if the system needs more capacity to complete a certain task. Such extra facilities can also be destroyed when the task has been completed, so system resources can be redirected to completing other tasks.

Much of the functionality provided by conventional languages, as well as concurrency, is implemented in occam 3. However, the many versions and revisions of occam resulted in a large and intricate language. The question can be asked whether this big language is still "intended to be the smallest language which is adequate for its purpose", as stated in [17], the first article about occam.

Stellenbosch University http://scholar.sun.ac.za

2.6 Model checking

Besides being a language for implementing embedded software, the language introduced in this thesis is intended as a language to assist verification techniques such as model checking. Therefore some attention has to be paid to existing model checkers and model checking techniques.

Spin [11] is one of the most powerful, well known and widely used model checkers available today. One of the advantages of using Spin is the language used to specify models for Spin model checking. Many verification packages use modelling languages or notations that are abstract and far removed from programming languages. In contrast Promela, the modelling language of Spin, resembles many implementation languages. The language is also based on CSP, but because it is used to express behaviour of systems implemented in other languages, some restrictions of CSP have not been implemented. For example, Promela allows buffered communication and sharing of variables between concurrently running processes. LF is intended as an implementation language, and can therefore be much more similar to CSP than Promela. Therefore LF has been influenced more by other CSP-based languages such as occam and Joyce, and Promela is not discussed in more detail.

Chapter 3 will discuss how the LF language was influenced by the goal to support model checking.

Stellenbosch University http://scholar.sun.ac.za

Chapter 3

The LF Language

Systems implemented in conventional languages can quickly become too intricate for model checking to be feasible. Abstracted models of such systems need to be built. However, such abstraction still needs to be applied by the same human intelligence that designs error-ridden systems. One solution to this problem may be to design an implementation language that supports model checking directly.

The experimental programming language LF described here is such a language. Synchronous communication and no shared data, as in CSP, simplifies model checking. Other mechanisms which enlarge the state space of a model, such as pointers and dynamic allocation of memory, have been eliminated. Points where processes can be pre-empted have been reduced — this limits the ways in which the execution of different processes can be interleaved, thus also reducing the state space.

The focus of this chapter is on the language and design goals for it. The form of the machine code generated supports model checking. This is covered in Chapter 4. Attention is given in this chapter to where and how coding errors can be avoided by the design of the language, and where and how the compiler can be used as tool to detect simple coding errors.

3.1 Previous work

An earlier version of LF was essentially an adaptation of Joyce for embedded work [21]. This included unsigned integer types, low-level operations for bit manipulation, typed pointers, and operations to write to hardware ports. A mechanism was also included to locate variables at

Stellenbosch University http://scholar.sun.ac.za

specified absolute memory addresses to communicate with memory-mapped devices.

An example, given in [21], of a program written in this first version of LF is listed below. It computes the tenth Fibonacci number by recursively instantiating `Fib` processes. On termination of the recursively called processes, the result is stored in the variable `i` in the `Caller` process (line 22). Instead of reference parameters, each process returns its result via a channel (line 11). New channels (referenced by ports `g` and `h`) are created dynamically (line 9) every time `Fib` creates two child instantiations of itself. The results of the children are then received back through `g` and `h` (line 10).

```
0   PROGRAM Ex012;
    TYPE CfuncVal = [f(UINT32)];
    PROCESS Fib(OUT func: CfuncVal; X : UINT32);
3   VAR
        IN g, IN h : CfuncVal;
        y, z : UINT32;
6   BEGIN
        IF x <= 1 THEN func ! f(x)
        ELSE
9           NEW(g); NEW(h); Fib(g, x-1); Fib(h, x-2);
            g ? f(y); h ? f(z);
            func ! f(y+z)
12      END
    END Fib;

15  PROCESS Caller;
    VAR
        IN result : CfuncVal;
18      i : UINT32;
    BEGIN
        NEW(result);
21      Fib(result, 10);
        result ? f(i)
    END Caller;
24
    BEGIN
        Caller
27  END Ex012.
```

Note that type `CfuncVal` is declared global, but variable `i` and port `result` are declared inside process `Caller`. This is because ports and variables cannot be declared globally.

Some areas where the language could be extended or improved were identified in [21, Chapter 5] and by programmers using the language:

1. The lack of procedures in LF was inconvenient. Because the caller of a procedure cannot continue execution while the procedure has not terminated, reference parameters can be implemented for procedures without allowing shared data. A similar construct in LF would lessen the amount of copying necessary in a system, increasing efficiency.

2. An entire implementation had to be written as one PROGRAM. This made it difficult to

Stellenbosch University http://scholar.sun.ac.za

package code for reuse, such as I/O libraries.

3. Constructs to support abstraction were rather limited.

4. The misuse of pointers in LF proved a serious obstacle to model checking efforts [4].

3.2 Design goals for the new version

The new version of LF described in this thesis is intended to support the development of reliable embedded software. The language should promote reliability and it should help programmers to create software for which computer-aided verification is practical. Since LF is intended as a tool to write embedded software, interfacing with hardware should be possible and natural. Efficiency is also a consideration for embedded software. Below, some design goals for the language are described in more detail.

3.2.1 Base the language on CSP

To simplify model checking, we have decided to base the language on CSP. Many techniques for checking CSP constructs have been developed. For example, the widely used Spin model checker can analyse CSP-like specifications for systems of realistic size and complexity.

3.2.2 Eliminate language features to make model checking feasible

To limit complexity and make model checking feasible, some features have been left out. For example, LF has no support for pointers, since indiscriminate use of pointers can enlarge the state space needed for model checking. The lack of pointers in occam supported this decision.

3.2.3 Safe programming practices

The language should encourage safe programming practices, and create opportunities for the compiler to indicate programming errors. For example, the language should be strongly typed; during assignments, parameter passing and interprocess communication the types of expressions being copied from should match the types of the variables being copied to.

Stellenbosch University http://scholar.sun.ac.za

3.2.4 Intuitive and easy to understand language

It should be easy to understand LF. An intuitive and clear language can limit errors and reduce development time. There should be no unnecessary exceptions to a general principle. To avoid confusion, a notation in LF which is also encountered in conventional languages should have *the same* meaning in LF as in conventional languages. Making the language as small as possible contributes to quick and easy understanding.

3.2.5 Small runtime system

Runtime support for embedded systems needs to be small and efficient, because of limited hardware resources. A run-time system executing CSP-like processes need only an efficient scheduler and efficient message passing mechanisms.

3.2.6 Low-level operations

Low-level operations to facilitate bit manipulation and communicate with peripherals are needed in the language. Interrupt handlers will also be written in LF as part of device drivers for embedded systems. Device drivers implemented as LF processes will allow the LF run-time system to be simplified and reduced in size. The CSP message passing paradigm provides simple interrupt handling functionality.

3.2.7 Context switching and interprocess communication in software

Most hardware architectures used for embedded systems do not include such highly efficient support for context switching and interprocess communication as do the INMOS transputer mentioned in section 2.2. Operations such as process creation and process termination are also more expensive on conventional hardware. Memory management hampers efficiency in any system, but an occam-like language would eliminate the need for dynamic memory allocation and deallocation (see Section 2.5.3).

LF is designed to execute on architectures with a scheduler in software. Since context switching will be less efficient than on the transputer, the language should encourage less context switching between processes than in occam. No hardware support for interprocess communication is assumed either; interprocess communication needs to be implemented in software by the run-time system.

Stellenbosch University http://scholar.sun.ac.za

3.3 Processes

All code in LF is encapsulated in processes. A process has its own private variables, and those variables cannot be shared with other concurrently running processes. Because of these disjoint data areas, race conditions that could occur with concurrent access to shared data are avoided. Processes contain sequences of commands. Message passing based on the CSP approach is used to exchange information between concurrent processes, and for synchronisation between such processes.

A process in LF has the following structure:

```
PROCESS Buffer;
    (* declarations *)
BEGIN
    (* commands *)
END Buffer;
```

The concept of a process is the main abstraction tool in LF, as is the case in CSP. Therefore LF follows the CSP example and allows process definitions to be nested, to allow different levels of abstraction.

3.3.1 Design considerations

Since the LF scheduler is implemented in software, context switches are relatively expensive. Where every occam command is a process, to be executed sequentially or in parallel, the concurrency model in LF is closer to Joyce to limit the number of context switches. Therefore an LF command is not a process — a process will always consist of zero or more sequentially executed commands.

3.4 Types, variables and constants

LF supports 32-bit, 16-bit and 8-bit signed and unsigned integers, 32-bit, 16-bit and 8-bit sets, a boolean and a character type, and static arrays and records. The language has no pointers, and process instantiation is the only way to create dynamic structures. Strong typing was considered essential to detect as many errors as possible at compile time. However, typecasts are provided so a programmer can explicitly override strong typing rules when necessary. Name equivalence of types, as defined in [22], is used.

Stellenbosch University http://scholar.sun.ac.za

Unsigned integer types are included because LF is intended for low-level embedded work. Such unsigned integers can be used to represent, for example, fields in protocol data packets. Set types provide a clean notation which can be used for bit manipulation. The current implementation of LF on the 80386 includes three differently sized set types because of the arrangement of data in bytes, words and doublewords.

Variables represent the private data of processes. They must be declared to be of a specific type; either pre-declared in the language, or user-defined. Constants are declared without type, and the compiler associates the constant with the smallest type which can represent the constant. For example, `UINT8` will be used if the constant is an integer greater than or equal to 0 and smaller than 256.

An example of a constant declaration below shows `BufferSize` declared as a constant with value `32` (it will therefore be a `UINT8`):

```
CONST
    BufferSize = 32;
```

Because the compiler decides the types of constants, the incompatibility of signed and unsigned types can cause problems — for example, the `BufferSize` constant might have been intended for use in signed expressions, even though strong typing prevents its use in signed expressions. A way to solve this problem would be to modify the language to let the user specify the type of constants.

Examples below show the declaration of user-defined types:

```
TYPE
    Name = ARRAY 32 OF CHAR;
    Payload = ARRAY 1024 OF CHAR;
    BufEntry = RECORD
        nm : Name;
        pld : Payload
    END ;
    Buffer = ARRAY BufferSize OF BufEntry;
```

`Name` and `Payload` are declared as array types of 32 and 1024 characters respectively. `BufEntry` is a record type, with fields `nm` (of type `Name`) and `pld` (of type `Payload`). Type `Buffer` is declared as an array type, with 32 elements of type `BufEntry`.

Some declarations of variables are shown below:

```
VAR
    bufferTotal, head, tail, rqNo : INT32;
```

Stellenbosch University http://scholar.sun.ac.za


```
buf : Buffer;
nm : Name;
pld : Payload;
```

Four 32-bit integers are declared: `bufferTotal`, `head`, `tail` and `rqNo` and the variables `buf`, `nm` and `pld` are declared to be of the user-defined types `Buffer`, `Name` and `Payload`, respectively.

3.4.1 Design considerations

Integer types

Because memory is abundant on most newer desktop systems, it can be an unnecessary and error-prone complication to support smaller and larger integers. A single, word-sized integer also makes it simpler to align variables on memory word-boundaries for efficient access.

However, for a language intended for low-level work, integer types of different sizes are useful. Having types for each different size of memory-mapped port makes it less cumbersome, for example, to communicate with peripherals via such ports. Another example where such types are useful is when protocols with predefined fields must be implemented.

Pointers

Pointers represent references to data stored in a shared memory pool (the heap). It was decided to avoid the problem of sharing dynamic data structures among concurrently executing processes by eliminating pointers. Although this may be the most controversial design decision taken in the design of LF, it certainly supports the goal of model checking as described in section 3.1.

3.5 Expressions, assignments and control structures

LF syntax for expressions is similar to that of Oberon [25]. Operator precedence is defined by the BNF definition of the language, given in Appendix A. Special operators different from the logical or arithmetic operators are needed to handle sets. Similar operators are implemented in Oberon: set union (`+`), set difference (`-`) and set intersection (`*`). The monadic minus sign

Stellenbosch University http://scholar.sun.ac.za

is used to obtain the complement of a set. The `IN` relation is used to determine whether a specific number is an element of a set.

Implicit type conversion in expressions is supported in a limited way. The table below lists all types on which such type conversion is performed. Every type is compatible with all types listed to the right of it in the same row.

Compatibility of types		
UINT32	UINT16	UINT8
INT32	INT16	INT8
SET32	SET16	SET8

Examples:

```
CONST
    BufferSize = 32;
    Sensor3 = 42;
VAR
    rqNo, head, tail : UINT32;
    bufNo : UINT8;
BEGIN
    ... rqNo = bufNo ... (* expression 1 *)
    ... head >= tail ... (* expression 2 *)
    ... 3 IN {3, 6, 9} ... (* expression 3 *)
```

The example above shows the relevant parts of three Boolean expressions. In expression 1, the 32-bit unsigned integer `rqNo` is compared for equality to the 8-bit unsigned integer `bufNo`; `bufNo` is implicitly converted to type `UINT32`. Expression 2 will be `TRUE` if `head` is greater than or equal to `tail`. Expression 3 is `TRUE` since 3 is an element of the set {`3, 6, 9`}.

Assignments in LF are similar to Oberon assignments.

Examples:

```
head := 0;
buffer[no].name := newName;
head := (head + 1) MOD BufferSize;
```

LF includes control structures similar to Oberon. Control structures implemented are `WHILE`, `CASE`, `IF` and `REPEAT`.

Stellenbosch University http://scholar.sun.ac.za

3.6 Process instantiation

As noted in section 3.1, procedures would be desirable in LF, since the ability to pass parameters by reference would decrease the amount of copying needed in the system. Processes which execute in sequence (similar to procedures) can be allowed to share data, since the data cannot be accessed by more than one process concurrently.

In LF, a process can execute concurrently with its instantiator, or it can leave the instantiator blocked while it completes execution (execute in sequence with the commands of its instantiator).

The keyword "CREATE" before a process name indicates that the process is to be executed concurrently with its creator, otherwise the creator is blocked (known as a *called* process). Information is passed from the instantiator to the *called* or *created* process via parameters. Parameters function as in Oberon. Processes are instantiated dynamically, and recursion is allowed, as the example below shows:

```
PROCESS Factorial(n : INT32; VAR ans : INT32);
VAR
    x : INT32;
BEGIN
    IF n = 1 THEN ans := 1
    ELSE
        Factorial(n-1, ans); ans := n*ans
    END
END Factorial;
```

The example shows a process which calculates the factorial of its first parameter by recursively instantiating itself, and returning the answer as a pass-by-reference second parameter.

3.6.1 Design considerations

Processes in LF are closer to Joyce agents than to occam processes. In occam any command can be explicitly specified to be executed sequentially or in parallel, but such fine-grained concurrency is unsuitable for most embedded hardware architectures because of factors such as inefficient context switching support (discussed in section 3.2). Joyce agents only execute concurrently, whereas LF processes can execute either concurrently, or as a procedure would have executed.

Stellenbosch University http://scholar.sun.ac.za

Dynamic process instantiation

In Chapter 2, the differing concurrency models of occam and Joyce were discussed. One of the observations was that recursion is not allowed in occam, allowing efficient machine code to be generated (section 2.2.5). In contrast, dynamic process instantiation in Joyce allows recursion to be implemented, but also introduces overhead into the language (section 2.3.1).

Because concurrency is not intended to be as fine-grained in LF as in occam, fewer context switches will typically happen in LF code and fewer processes will be created. Therefore the overhead of dynamic process instantiation was deemed acceptable.

Called processes

In conventional procedural languages, a called procedure must complete before the calling procedure can proceed. In concurrent languages such as Joyce (section 2.3), processes share many properties with procedures: both receive information via parameters when instantiated, both own local variables and both consist of a number of commands. However, in Joyce two processes instantiated one after another can potentially execute in parallel. In occam, where every command is viewed as a separate process, a programmer can specify whether to execute processes in sequence or in parallel.

LF processes are similar to processes in Joyce. It was decided, however, to give the programmer the additional ability to instantiate a child process and have it *execute as a procedure would.* Therefore LF retains the procedure call semantics familiar to most programmers. In fact, the process being called and the process calling still exist concurrently, but the callee completes execution while the caller remains blocked.

A typical application of the *called* process is to put a sequence of often repeated commands in a separate process. This separate process can then operate on a big data structure defined in the caller without passing the structure to the *called* process — something that would not be possible in Joyce. For example, the process `CheckCoords` below checks that every X and Y coordinate in an array of coordinates is within a specified range. The array of coordinates is defined in the scope of the encapsulating process `MaintainCoords`. If either the X or Y coordinate is not within range, both coordinates are changed to `0`.

```
PROCESS MaintainCoords(ch : Chan);
CONST
    UpperBound = 120;
    LowerBound = -60;
TYPE
```

Stellenbosch University http://scholar.sun.ac.za


```
    Coords = RECORD x, y : INT32 END ;
    CoordsArray = ARRAY 5000 OF Coords;
VAR
    arr : CoordsArray;
    i : INT32;

    PROCESS CheckCoords(i : INT32);
    BEGIN
        IF (arr[i].x > UpperBound) OR (LowerBound > arr[i].x)
        OR (arr[i].y > UpperBound) OR (LowerBound > arr[i].y) THEN
            arr[i].x := 0; arr[i].y := 0
        END
    END CheckCoords;

BEGIN
    (* ... *)
    i := 0;
    WHILE i < 5000 DO
        CheckCoords(i);
        i := i + 1
    END ;
```

The command to *call* a process has the same syntax as a procedure instantiation in Pascal, to suggest to the programmer he or she can expect procedure call semantics. To *create* a process, the CREATE keyword is used to suggest that this command differs from *calling* a process.

Variables can be shared between a *called* process and its caller, since the caller will remain blocked while the callee executes and variables will not be accessed concurrently. This is similar to processes in occam which are executed as part of the SEQ construct. However, because there is no dynamic process instantiation in occam, addresses of all variables can be resolved at compile time. In LF, static links from nested processes to the processes encapsulating them are maintained to support addressing of global variables. These links will be similar to static links in procedural languages.

An aspect that makes Joyce inconvenient for large software projects, is that no analogues for modules exist. These abstraction facilities are desirable to enhance the maintainability and understandability of large implementations. To implement embedded systems, more support for system-level programming such as device drivers has to be included in the language. Examples of such facilities are bit manipulation operators, facilities to accurately determine passage of time, and facilities for processing interrupts.

Processes are generalised further in the sense that a process can declare reference parameters, and return a result. Such processes are restricted to being *called*, because reference parameters amounts to shared data, and a caller will need the result of a function before continuing execution. This is checked by the compiler.

Stellenbosch University http://scholar.sun.ac.za

A (somewhat contrived) example of a process that illustrates reference parameters and a result is given below:

```
CONST Success = TRUE; Failure = FALSE;
PROCESS Add(VAR argl, arg2 : INT32) : BOOLEAN;
BEGIN
    argl := argl + arg2
    RETURN Success
END Add;
```

Below is illustrated how such a process is *called*:

```
VAR
    answer : INT32;
    result : BOOLEAN;
BEGIN
    (* ... *)
    answer := 5;
    result := Add(answer, 6)
```

3.7 Interprocess communication

Synchronous communication is the only means by which concurrently running processes can exchange information in LF. The mechanisms, declarations and commands used to implement communication in LF are discussed below.

3.7.1 Channels and ports

The LF runtime system uses data structures called *channels* to implement synchronous communication. Channels are allocated and deallocated dynamically by the runtime system. A process wanting to communicate via a channel can do so by using a *port*, which is a reference the process has to a channel. The first process needing a channel declares a port, and allocates the corresponding channel by executing the NEW command. The run-time system then creates a channel, and stores a reference to the channel in the port.

Ports have special types called alphabet types, similar to port types in Joyce. The alphabet contains a number of symbols, each describing a different class of message. A symbol sent over a channel can be accompanied by one or more values. Special symbols called *signals* are not accompanied by any data, and are intended to notify processes of the occurrence of important events.

Stellenbosch University http://scholar.sun.ac.za

The example below shows the declaration of an alphabet type `PrinterOp` with three symbols, `Startup`, `Print` and `PrintJobs`. `Startup` is a signal, so no data will be transmitted with it. `Print` indicates that an `INT32` value and a value of type `PayloadType` are to be transmitted. `PrintJobs` is accompanied by an `INT32` value. After the alphabet type has been declared, a port of that type is declared and `NEW`ed.

```
TYPE
    PayloadType = ARRAY 1024 OF CHAR;
    PrinterOp = [Startup, Print(INT32, PayloadType),
        PrintJobs(INT32)];
PORT
    op : PrinterOp;
BEGIN
    NEW(op);
```

3.7.2 Sending and receiving messages

To send or receive a message, the programmer supplies

1. a reference to a channel,
2. either the values to send, or the variables in which to store the values received, and
3. an alphabet symbol associated with the message.

A process receiving a certain symbol can only communicate with a process sending the same symbol, and *vice versa*. In Joyce, channels can only transmit or receive a single value per alphabet symbol. In LF this single value has been replaced by a comma-delimited list of values, similar to a parameter list in a procedure call. An example of sending and receiving in a simple print server is shown below.

```
TYPE
    PayloadType = ARRAY 1024 OF CHAR;
    PrinterOp = [Startup, Print(INT32, PayloadType),
        PrintJobs(INT32)];
PORT
    op : PrinterOp;
VAR
    id, jobs: INT32;
    pld : PayloadType;
BEGIN
    NEW(op);
    op ? Print(id, pld);
    jobs := jobs + 1;
    op ! PrintJobs(jobs);
```

Stellenbosch University http://scholar.sun.ac.za

The data which accompany each symbol are specified between brackets after every symbol name. The process containing the commands above receives a `Print` symbol accompanied by an ID number and the payload (data to be printed). Later, the process informs another process how many print jobs are currently handled, by sending the `PrintJobs` symbol, accompanied by the number of print jobs.

Processes that want to send or receive are blocked until communication partners are found for them. References to blocked processes are inserted into queues in the channel. There is one queue per symbol and the symbol number is used as an index to identify the right queue. A diagram of how processes are queued in a channel is given in Figure 1.

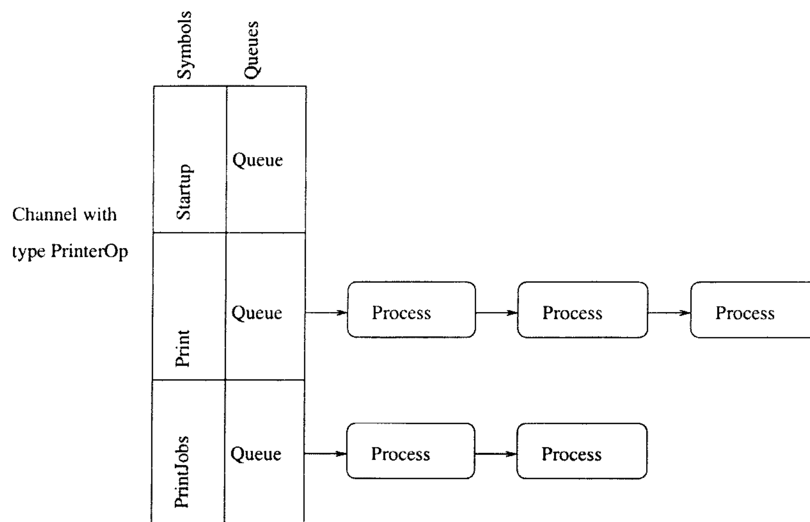

Figure 1: The queueing of processes in a channel of type `PrinterOp`

The drawing represents a channel with alphabet type `PrinterOp`. Three processes are waiting to send the symbol `Print`, and two processes are waiting to receive the symbol `PrintJobs`. Processes can be queued waiting to send or receive on each of the symbols. If a process attempts to send on a channel and a matching receiver is found in the queue, the run-time system copies the data from the sender to the receiver, and unblocks both processes. The same happens when a process attempts to receive on a channel and a matching sender is found in the queue.

Each symbol has only one queue, which can contain either senders or receivers, but not both (senders *and* receivers). This is because some processes cannot be waiting to send while other processes are waiting to receive the same symbol on the same channel. In such a case the senders and receivers will have been matched up one-to-one before both queues are occupied.

Stellenbosch University http://scholar.sun.ac.za

To simplify interfaces between processes, it was decided to allow concurrently running processes to share ports, similar to Joyce. Ports can be shared between, for example, the process defining the port and all processes nested in that defining process. Nested processes accessing global variables can only be *called*, but nested processes accessing global ports can still run concurrently with the parent. An example of where this feature is convenient is shown below:

```
PROCESS Container(rq1, rq2, rq3, rq4, rq5 : PortType);

    PROCESS Auxiliary;
    VAR
        x : INT32;
    BEGIN
        rq3 ? msg(x)
        (* ... *)
    END Auxiliary;

BEGIN
    Auxiliary
    (* ... *)
END Container;
```

The `Auxiliary` process automatically has access to all ports in its parent scope.

3.7.3 Design considerations

The IPC model in LF differs from that used in conventional operating systems such as Unix. The design of the IPC model influenced other aspects of the LF language, and was also influenced by other design issues in the language. Below, IPC in LF is compared to that of more conventional systems, and the influences and consequences of the design decisions are discussed.

Direct versus indirect naming

To communicate, processes need to specify their communication partners. Therefore, some way is needed to identify or *name* processes. This naming can either be *direct* or *indirect.* With direct naming, the process supplies the name of the communication partner as part of the communication command. Direct naming is used in the version of CSP discussed in [10] (which languages such as occam are based on), but this convention cannot be used in our environment, since LF process instantiations are anonymous. (Anonymous process instantiation simplifies, for example, the implementation of dynamic and recursive process instantiation, since no names have to be given to processes instantiated at run-time.) With indirect naming, mailboxes or channels are used for communication, so both the sender and

Stellenbosch University http://scholar.sun.ac.za

the receiver specify the channel to/from which communication is expected, with the channel accessible to both processes. This scheme is more suited to the LF environment, and is the scheme which is used. This is discussed in more detail in [21].

Channel ownership

When conventional operating systems use indirect naming as discussed above, messages are transmitted via *mailboxes* or *ports* [19, Chapter 4]. Either the run-time system or the processes can own such mailboxes. If mailboxes are owned by processes, only the owner processes may receive over the mailboxes. Therefore messages are transmitted in only one direction. If the operating system owns mailboxes, the reception privilege initially belongs to the process that allocated the mailbox. However, this privilege can be transferred to other processes.

A different concept, inherited from Joyce, is used in LF. The directions in which messages are sent are not determined by ownership. Mailboxes are replaced by bi-directional channels owned by processes, and the owner has the responsibility to create channels. This is done by executing the `NEW` command. Whenever a process terminates, the runtime system destroys the channels owned by that process.

Processes store references to channels in ports, which are subject to Oberon-like scoping rules. Therefore a port can be seen only inside the process that declared it, by the declaring process itself and all nested processes. Additionally, nested processes can themselves only be seen by the process in which they are defined, and a parent process can only terminate once all its children terminate. Therefore there is no risk that a process might use a port of a process which has already terminated. Using an uninitialised port for communication (referencing a channel not `NEW`ed yet) results in a run-time exception.

Dynamic channel allocation and shared channels

Channel limitations in occam imply that a server has to have a channel for every one of its client processes. This severe restriction allows for efficient embedded implementations. It does complicate maintaining and extending occam software. However, the designers apparently deemed this an acceptable trade-off for embedded software, which is normally smaller, non-portable and implemented in a low-level language.

LF supports dynamic process instantiation similar to Joyce, as stated in section 3.6.1. Dynamically created processes are of little use if the communication network cannot be extended

Stellenbosch University http://scholar.sun.ac.za

to include these new processes. For example, consider a parent process in a loop, continually instantiating new children. New channels have to be dynamically allocated along with the new children if the children are to be used in the LF program.

Channel sharing simplifies communication between dynamically created processes. For example, a server process can accept requests from any number of client processes via a single input channel shared between all processes. Channel sharing is expected to work well for lightly used channels, as Brinch-Hansen argues in [8]. However, it is possible that client processes can be starved of access to a heavily used server process accessed via channels. In this case, several servers or several channels to the server can be used. When multiple senders and receivers on the same channel communicate messages of the same type, the run-time system decides how to match up senders and receivers.

Unfortunately channel sharing complicates the implementation of channels and introduces overhead. For example, it is possible for more than one process to be blocked on a channel. Therefore it is necessary to implement queues for channels as shown in Figure 1, page 34; because messages are matched up according to alphabet symbols, a queue is needed for each alphabet symbol that can be sent over the channel. However, if channel sharing were not supported, as in occam, some queueing mechanism for client-server architectures would have had to be implemented by the user.

Type checking of messages

The typing of messages also offers benefits. As Brinch-Hansen stated in [8], type errors in interprocess communication were frequently reported by the Joyce compiler. This is an example where the compiler can detect errors that would otherwise be difficult to trace. The fact that different classes of messages, denoted by alphabet symbols, can be sent, simplifies the interfaces between processes. The absence of this feature would necessitate large networks of channels between processes.

Port sharing

Ports share several qualities with variables. They are declared in the same fashion as variables, and can be seen as references — they each contain a reference to a channel. Yet ports can only be changed by a `NEW` command, and ports are typically only `NEW`ed once. Therefore ports do not need to be protected from concurrent access. Also, the communication over channels is synchronous, so simultaneous accesses to channels are serialised. Since sharing of

Stellenbosch University http://scholar.sun.ac.za

ports cannot lead to race conditions, sharing of ports is deemed acceptable, and supported in LF. Consequently, a nested process has unrestricted access to a port declared in an encapsulating process. As stated in section 3.6.1, a nested process accessing a variable defined in an encapsulating process can only be *called*, not *created*.

Benefits of additional restrictions on the NEW command

Although processes cannot interfere with one another by changing the value of a port, a message can be lost by sharing a port. When two processes share a port, one process can send a message over a port, and either one of these processes can then re-initialise the port before the message is received.

It is possible to let the compiler restrict where and when the NEW initialisation command can be used to prevent such errors from occurring. One could, for example, restrict the NEW command to be used only to initialise ports declared in the local scope of the process. One could also perform flow analysis on the code of a process to ensure that a port is only initialised once. However, these kinds of restrictions might lead to rejection of syntactically correct code — an example of code which is correct, but might be rejected by a compiler because of restrictions as discussed, is shown below:

```
PROCESS Server(requestType : INT32);
PORT
    p : ChanType;

    PROCESS Aux1;
    BEGIN
        NEW(p)
        (* ... *)
    END Aux1;

    PROCESS Aux2;
    BEGIN
        NEW(p)
        (* ... *)
    END Aux2;

BEGIN
    IF requestType = 1 THEN Aux1
    ELSIF requestType = 2 THEN Aux2
    END
END Server;
```

It might have been preferable if port p was initialised in process Server instead of in Aux1 and Aux2, but the code listed above is still syntactically correct. It was therefore decided to not implement these checks and restrictions. Instead of such restrictions, model checking of implementations in LF can point out where messages can be lost because of repeated NEW

Stellenbosch University http://scholar.sun.ac.za

commands with the same port.

3.8 A construct for nondeterminism

The LF equivalent for the CSP general choice operator "□" is known as the SELECT command.

```
SELECT
    WHEN ch1 ? msg1(x) THEN y := 1
    WHEN ch2 ! msg2(x) THEN y := 2
END
```

Several alternative command sequences are given, each denoted by a WHEN keyword. The first command, following each WHEN keyword, is known as a guard. In the example above the guards are `ch1 ? msg1(x)` and `ch2 ! msg2(x)`. The rest of the command sequence (`y := 1` or `y := 2`) follows the THEN keyword. A SELECT blocks until a guard becomes executable, then executes the guard and the rest of the command sequence to which the guard belongs. If more than one guard is TRUE, one will be selected non-deterministically, and its command sequence executed.

As with the Joyce `poll` command (section 2.3.3), a guard consists of a simple communication command (send or receive), and an optional boolean expression. If a guard does not contain a boolean expression, the guard becomes TRUE if the communication is possible. If a boolean expression is present, the communication should be possible and the boolean expression should evaluate to TRUE.

The example below shows a more complicated SELECT command.

```
SELECT
WHEN op ? Print(id, payload) & jobs > 0 THEN
    (* print *)
WHEN op ! PrintJobs(jobs) THEN
    (* ... *)
END
```

The first guard contains a communication and a boolean expression. It is TRUE if a `Print` message is available, and `jobs` is greater than 0. The second guard is TRUE if a `PrintJobs` symbol can be sent.

The SELECT in LF also implements a conditional receive similar to that described in section 2.3.3 for the Joyce `poll`. To illustrate this, the first guard in the example above can be modified to reference a value to be received:

Stellenbosch University http://scholar.sun.ac.za


```
WHEN op ? Print(id, payload) & id = 5 THEN
    (* ... *)
```

The value of `id` in the boolean expression `id = 5` will be changed if the communication takes place. However, since the boolean expression is part of the guard, the value of `id` will be the value *to be received*. This gives a process the ability to test the contents of a message before removing it from the channel and receiving it.

3.8.1 Design considerations

The interprocess communication in LF is similar to that in Joyce [8]. Channels can be shared, and typed messages are classified according to alphabet symbols. Because of this similarity, the `SELECT` in LF is similar to the Joyce `poll`. Some implications of the construct are discussed below.

The need for nondeterminism in CSP-based languages

All of the CSP-based languages discussed in Chapter 2 implemented a construct similar to the occam `ALT` or the LF `SELECT`, except one. With Planet, Crookes and Elder purposefully omitted such a construct to determine how necessary it really is [7]. They found that the omission did not influence the implementation of deterministic applications such as compilers. However, for applications such as operating systems which involve nondeterminism, "severe difficulties were encountered".

Planet allowed nested processes access to the variables of their parents, so such applications were implemented with nested processes communicating via shared variables and synchronisation queues. The order in which processes were scheduled on such a system (not known at compile time) can then be used to introduce nondeterminism into the control flow of the system. However, to ensure exclusive access to shared variables, a 'run-to-suspension' process scheduling strategy became necessary, and real-time response capabilities of the system were negatively affected.

Therefore, a programmer can use shared variables (together with a run-time scheduler) for nondeterminism if no non-deterministic construct is available. However, the occam `ALT` construct *explicitly* specifies several alternate execution paths. In contrast, it can be easy to overlook an implied (and possibly erroneous) execution path using shared variables for

Stellenbosch University http://scholar.sun.ac.za

nondeterminism. To limit possible paths for scheduling, synchronisation primitives such as semaphores or monitors are needed. Such mechanisms are superfluous in occam or CSP.

Conditional receive guard

As described earlier in this section, a guard can conditionally receive a message based on the contents of that message. This is made desirable by the sharing of channels in Joyce. Suppose several processes are listening for a message on a channel. The receiver of the message is non-deterministically selected by the run-time system. However, the message might not be intended for the selected receiver. The conditional receive in Joyce allows an agent to examine a message before receiving it, and decide whether or not to receive the message.

Implementation of this feature proved cumbersome, and the addressing of data to be received incurs overhead. Yet, if this feature was not implemented once and provided by the LF system, users would have to implement it repeatedly, by receiving a message, testing it and re-sending it if it was not the correct message.

The designers of occam focused on providing the minimum functionality and having a simple, efficient run-time system above all else. The run-time system for LF implements more functionality which complicates implementations of the compiler and run-time system, and incurs more overhead. However, it relieves the programmer of implementing some often-needed features. It also eliminates the potential errors which the programmer could have made when implementing such features.

Output guards in a `SELECT` command

Neither the occam `ALT` construct nor the "□" operator in the original version of CSP [10] allows an output command as part of a guard. In contrast, guards in an LF `SELECT` command may contain output commands. However, two `SELECT` commands may not synchronise. This is enforced by only letting a `SELECT` guard communicate with a matching simple communication command, similar to the way in which Joyce `poll` guards are prevented to communicate with other `poll` guards.

To reason about guards in an `SELECT` command which *can* communicate with guards in another `SELECT`, suppose a guard G_i attempts to transmit over a channel C_i. Every other guard G_j in every other `SELECT` which attempts to receive over channel C_i, needs to be examined to determine if guard G_i is `TRUE`. Also remember that a guard can include a boolean

Stellenbosch University http://scholar.sun.ac.za

expression as well as a communication. If a communication partner for a communication in the guard has been found, a boolean expression might potentially have to be evaluated as well. Therefore the amount of processing involved if several processes are simultaneously blocked on a SELECT on the same channel is considerable.

A use for SELECT statements with matching guards could not be envisioned in either [21] or by this author. Therefore guards in one SELECT cannot communicate with guards in another SELECT in LF. References to several discussions on this subject are listed in [21].

Because LF allows output guards, the input and output of a ring of processes can be implemented more symmetrically. To illustrate this, an example of an occam ALT and an LF SELECT which perform the same function in a ring of processes are contrasted below. The occam example is listed first:

```
0  ALT
1    in ? buf[tail]
2      -- do housekeeping
3    prompt ? giveBufferedValue -- can only poll input values
4      out ! buf[head]
5      -- do housekeeping
```

On line 1 a value to be buffered is received over channel in. However, to output a buffered value, a giveBufferedValue message has to be received by the buffer first (line 3), before a value can be output (line 4) over channel out. The LF equivalent of the occam code listed above is:

```
SELECT
WHEN in ? val(buf[tail]) THEN
    (* do housekeeping *)
WHEN out ! val(buf[head]) & (head # tail) THEN
    (* do housekeeping *)
```

The first WHEN guard receives a value to be buffered, and the second WHEN guard unbuffers a value and sends it over channel out. The extra input in line 3 of the occam example is eliminated.

Stellenbosch University http://scholar.sun.ac.za

3.9 Modules

Modules in LF support separate compilation and abstraction. All static information defined in a module, such as process definitions, constants and types, as well as ports can be exported (made visible to other modules importing the module). Exported variables are not allowed, since processes in other modules could share such data.

An example of a module is shown below. The module implements print buffers. To export an entity, it is marked with an asterisk. The entities exported include a constant (`BufferSize`), two types (`PayloadType` and `PrinterOp`) and a port (`op`). No processes are exported in this example; the port `op` is the interface the module supplies to the rest of the system.

```
MODULE PrintBuffers;
(* This module implements functionality for several print buffers, all
    listening on a shared channel for buffering requests. Every buffer
    corresponds to a printer coupled to the system. Functionality to
    print is implemented by module "Printer", exporting a channel
    which will accept messages specifying print jobs as soon as the
    printer is ready to print. *)
IMPORT Printer (* module implementing printer functionality *);
CONST
    BufferSize* = 32;
TYPE
    PayloadType* = ARRAY 1024 OF CHAR;
    PrinterOp* = [Startup, Print(INT32, PayloadType),
        PrintJobs(INT32)];
PORT
    op* : PrinterOp;
VAR
(* total number of buffer processes maintained on system;
    corresponds to number of printers coupled to system *)
    bufferTotal : INT32;
(* concurrent process used to maintain single print buffer *)
PROCESS Buffer(bufNo : INT32; op : PrinterOp);
    (* call-only process used to insert entry at position 'no' *)
    PROCESS InsertEntry(newId : INT32; VAR newPld : PayloadType);
    BEGIN
        buffer[no].id := newId;
        buffer[no].payload := newPld
    END InsertEntry;
    (* ... *)
BEGIN
    (* ... *)
END Buffer;

BEGIN
    NEW(op);
    CREATE Buffer(0, op); CREATE Buffer(1, op); CREATE Buffer(2, op)
END PrintBuffers.
```

Note the BEGIN..END at the end of the example, representing the module "body". This area is used by the programmer to initialise the port and *create* the concurrently running processes. After initialisation, the module activation record (containing the exported port which is the only interface of the module to the system) stays allocated, even though no more

Stellenbosch University http://scholar.sun.ac.za

code is executed. This is because other processes will use the exported port after the module has finished execution.

3.9.1 Design considerations

One of the weaknesses of the original LF discussed in section 3.1 was that there was no tool for modularisation and code reuse. The PROGRAM construct in the original LF was an inert container construct for processes and type definitions — nothing could be exported or imported from it. Therefore LF had no way to implement libraries of processes, for example. A PROGRAM also contained no data, since all processes in the first LF prototype executed concurrently, and global data would have been shared between all processes in the PROGRAM. An initial process which accepted no parameters was always needed to initialise channels and start up other processes which needed parameters. The Caller process in the example in section 3.1, partially reprinted below, is such an initial process. Its only function is to contain and initialise the channel and actual parameter needed by process Fib.

```
PROGRAM Ex012;
TYPE CfuncVal = [f(UINT32)];
PROCESS Fib(OUT func: CfuncVal; x : UINT32);
    (* ... *)
END Fib;

PROCESS Caller;
VAR
    IN result : CfuncVal;
    i : UINT32;
BEGIN
    NEW(result);
    Fib(result, 10);
    result ? f(i)
END Caller;

BEGIN
    Caller
END Ex012.
```

It seemed that an encapsulating process would be a better tool for modularisation than the PROGRAM construct, since it would contain its own data, and could perform initialisation of its own ports and variables, thereby removing the need for an additional process similar to Caller in the example above. However, an interface between the container process and the rest of the system is needed if the container process is to implement modularisation of implementations. It is not possible to define such an interface *inside* the process and make it visible outside the process without changing the Oberon-like scoping rules in LF. However, the scoping provides security to the LF language. For example, it prevents processes from referencing data or using channels belonging to processes which have already terminated.

Stellenbosch University http://scholar.sun.ac.za

It was therefore decided to create a specialised construct for modularisation similar to a 'module' in Oberon [23]. A module in LF is a specialisation of a process — it can be defined to contain data (variables and ports) and its own constant, type and process definitions. Ports, types and constants do not represent data that can be shared between processes, and can therefore be *exported* or made visible in other modules. Variables represent data that can be shared, and may not be exported. Likewise, processes which do not reference global data may be exported, allowing the user to create libraries of often-used processes.

The ports of modules can be exported and used at any time by any process in the implementation. Therefore the channels which those ports refer to must stay allocated while the implementation is running. In section 3.7.3 it is stated that a channel is deallocated once the process which created that channel terminates. Therefore, to maintain consistency, modules should never terminate.

In Oberon, a module is loaded on demand and stays resident in memory until it is unloaded by the user. An LF module is created when the implementation starts up, and does not terminate when it has executed all its commands. Rather, it is blocked permanently, so it does not need to be scheduled again, yet the space for its data (variables and ports) stays allocated while the implementation is executing. In contrast with Oberon, there is no way to deallocate modules. However, LF modules cannot be instantiated dynamically, so the amount of memory used for modules will not increase while the implementation executes.

3.10 Constructs to facilitate low-level programming

Since LF is intended for the implementation of embedded software, low-level operations are needed. Features in LF to support such operations are given below.

3.10.1 Communicating with peripherals via memory

Memory-mapped devices, such as video displays, are mapped into the address space at specific addresses to obtain information or commands. LF provides for the declaration of variables at specific addresses to interface with such devices.

The example below shows a memory-mapped text video display which is addressed by a matrix placed at the right memory locations:

```
TYPE
```

Stellenbosch University http://scholar.sun.ac.za


```
    LineDisplay = ARRAY 60 OF CHAR;
    Display = ARRAY 38 OF LineDisplay;
VAR
    disp : Display AT 08000H;
```

I/O-space ports hardware ports exist on certain architectures. These are not to be confused with LF ports which reference channels and serve to interface between peripheral devices and software. Such ports cannot be addressed with the AT construct, because writing to such ports will not send information to the desired device. On Intel architectures, data is sent to the device with a specific OUT instruction, and received from the device with an IN instruction. The LF implementation discussed in this thesis, written for the Intel 80386 architecture, predefines the PORTIN and PORTOUT processes, which generate these instructions.

The example below shows an 8-bit value written to a port to communicate with the timer device on the system.

```
VAR
    timerLatch : INT8;
BEGIN
    PORTOUT(TimerLatchAddress, timerLatch);
```

The full list of intrinsic (predefined) processes is given in Appendix B.

3.10.2 Time measurement in LF

The LF runtime system maintains a clock value that is updated in timer ticks; the duration of a timer tick is system-dependent. To obtain this clock value, an intrinsic channel is provided, similar to the TIME channel in occam [13, Chapter 3]. A process receiving the NOW symbol over this channel will receive a signed 32-bit value indicating the number of ticks that have elapsed since either system start-up, or since the last wrap-around of the clock. A process sending the TIMEOUT symbol over the TIME channel is blocked; the process will only be scheduled again once the number of timer ticks in the TIMEOUT message have elapsed since the sending of the message. The functioning of the timer channel is illustrated below:

```
(* get the number of timer ticks elapsed *)
TIME ? NOW(printTime);
(* timeout *)
TIME ! TIMEOUT(200);
```

The NOW message from the TIME port can be used as a guard inside a SELECT. However, the communication will always be possible, so the NOW guard will always be executable. However,

Stellenbosch University http://scholar.sun.ac.za

the TIMEOUT message over the TIME port cannot be used inside a SELECT; a process will necessarily remain unscheduled (so no polling of SELECT guards can take place) for the amount of timer ticks specified in the message.

3.11 The SYSTEM module

The SYSTEM module defines the interface between the hardware and LF processes. It is dependent on the implementation of the run-time system, and supplies an interface by which device drivers can process interrupts, and gain access to system-dependent services.

3.11.1 Interrupts

To interface with hardware, some way is needed to process interrupts. The SYSTEM module exports a number of ports intended for interrupts. These ports are not initialised with the normal *call* to NEW. Rather, a different process, ASSIGN is *called* inside the SYSTEM module. This intrinsic process assigns each port to a specific interrupt. If a port is assigned to an interrupt, and the interrupt occurs, the run-time system will send a signal over the corresponding channel.

An outline of a keyboard driver below illustrates the use of interrupt channels. The interrupt handler process (Handler) is notified of interrupts through the keyboard channel. After servicing of the interrupt, Handler sends the keypress to Buffer via the keypressed channel. The buffer stores all characters until the buffer is full, and then discards characters. The rest of the system can obtain keypresses from the buffer process by receiving from the exported keys channel.

```
MODULE KeyboardDriver;
IMPORT SYSTEM;

TYPE
    KeyRequestChan* = [Buffered(CHAR)];
    KeyPressedChan = [Pressed(CHAR)];
VAR
    keys* : KeyRequestChan;
    keypressed : KeyPressedChan;

PROCESS Buffer( out : KeyRequestChan; in : KeyPressedChan );
(* implements a buffer for keyboard input.
    - processes attempting to receive are blocked if the buffer is
        empty
    - characters added to a full buffer are discarded without
        blocking the sending process *)
BEGIN
    WHILE TRUE DO
```

Stellenbosch University http://scholar.sun.ac.za


```
        SELECT
        WHEN in ? Pressed( keyIn ) THEN
            IF count < buffermax THEN (* add keyIn to buffer *) END
        WHEN out ! Buffered( keyOut ) & count > 0 THEN
            (* remove value from buffer and initialise keyOut to
            next value to be removed from buffer *)
        END
    END
END Buffer;

PROCESS Handler( keyboard : SYSTEM.Interrupt; buffer : KeyPressedChan );
(* interrupt handler for keyboard interrupts *)
CONST
    KeyPortA = 0A4H;
VAR
    key, scanCode : CHAR;
    PROCESS Translation(scanCode : CHAR) : CHAR;
    BEGIN
        (* ... *)
    END ;
BEGIN
    WHILE TRUE DO
        keyboard ? interrupt;
        PORTIN(KeyPortA, scanCode); (* read character *)
        key := Translation(scanCode);
        buffer ! Pressed(key)
    END
END Handler;

BEGIN
    NEW(keys); NEW(keypressed);
    CREATE Buffer(keyboard, keypressed);
    CREATE Handler( SYSTEM.Int1, keypressed );
END KeyboardDriver;
```

The buffer is implemented in a separate process from the interrupt handler so the interrupt signal is not received in a SELECT. If an interrupt signal is received by a SELECT guard, the process waiting for an interrupt would not be waiting to receive on the interrupt channel, but would be testing several communication alternatives. In such a case, it is not guaranteed that the interrupt will be processed once the signal is sent; another guard evaluated before the interrupt guard may be TRUE. In this way another interrupt of the same kind could occur before the first interrupt is processed. Therefore, to ensure timely processing of an interrupt, the process blocks on receive on the interrupt channel.

3.12 Scoping and concurrent access

Although LF adheres to the principle of no shared data between concurrent processes, it also allows nesting of processes. Programmers used to Oberon-like languages would expect nested processes to be able to reference the variables and ports declared in the encapsulating processes. LF does follow the scoping rules of Oberon; however, if a process references a variable defined outside its scope, the compiler limits the programmer to only *calling* such a

Stellenbosch University http://scholar.sun.ac.za

process. The nested process can in this case not run concurrently with the process defining the variable.

This is one way in which compile time checking can ensure that concurrent processes do not share data, while the scoping rules remain as an Oberon programmer would expect it to be. Because data can now be shared, unnecessary copying of data via message passing and parameters passed by value can be avoided. Sharing of data between sequentially executing processes makes it possible to allow reference parameters in the language without violating the assumption that there is no data shared between concurrently running processes.

A way to further relax restrictions on the programmer while still maintaining disjoint memory spaces for processes would be to enforce restrictions on individual variables instead of processes. This would be closer to the usage rules for variables and channels in occam, discussed in section 2.2.7. The compiler could check that, if a nested process shares a variable with an encapsulating process, the encapsulating process does not reference the shared variable after *creating* the nested process. This would enhance concurrency with no negative effects except complication of the compiler.

3.13 Security claims

Data race conditions are caused by processes erroneously affecting changes in the data of other processes. It is claimed that LF avoids these race conditions by disallowing shared memory altogether. Interference because of shared data can occur in the following guises:

- **Stack overflows** can cause overwriting of data if the system does not perform run-time stack checking. If stack checking is performed, execution is aborted when a stack overflow is trapped — an undesirable occurrence on most embedded systems. LF processes do not need run-time stacks, because:
  - no procedure call stack is needed. Even though *called* processes are really procedures, the memory for these processes are maintained in a linked list of process records by the kernel, together with the process records for concurrently running processes. The memory of terminated processes are reclaimed and reused for new processes. Of course, unbounded recursion will still exhaust free memory and cause the system to trap, but no data can be overwritten.
  - expressions are calculated in registers for efficiency

Stellenbosch University http://scholar.sun.ac.za

- **Shared memory**, which is eliminated from the LF environment:
  - global variables shared between concurrently executing processes are disallowed,
  - reference parameters cannot be passed to a process which will execute concurrently with the caller (causing the caller and callee to share the data referenced),
  - pointers are not supported.
- LF has the **AT construct** to declare variables at specific addresses. This feature is intended for limited use by device driver programmers familiar with the LF environment and memory layout. It can therefore be abused to break the security offered by the environment, by declaring two variables in different processes to be located in the same memory area.
- **Array indexing** which exceeds the bounds of the array. The compiler inserts index checks for all array indexing operations. These are, however, expensive run-time checks. When it is possible to verify validity of indexing for a given array, run-time index checking can be eliminated.
- **References to uninitialised ports** (not NEWed or ASSIGNed) are trapped with run-time checks; this is made possible by the run-time system initialising all process data to 0. These errors could also be detected by model checking.

Other security claims:

- Overflows on arithmetic operations are trapped by run-time checks.
- LF is a strongly typed language. Therefore the compiler can perform compile time checks to ensure that only compatible types can be used in expressions and assignments, and actual parameters are compatible with formal parameters.

When a run-time check fails, a machine exception is generated, and default behaviour of the system is to halt. As mentioned in section 3.2, it is possible to write an exception handler to trap an exception, but no special language constructs to recover from exceptions have been included as yet. However such features, if included, will provide for the design of fault-tolerant systems.

Stellenbosch University http://scholar.sun.ac.za

3.14 Some LF examples

To illustrate how the language can be used, some LF examples are given. Since the language is similar to Joyce, some examples from [8] have been rewritten in LF. These are listed first (section 3.14.1 to section 3.14.4).

3.14.1 Generate a bounded stream

This process generates an arithmetic progression $a_0, a_1, \ldots, a_{n-1}$ where $a_i = a + i * b$.

```
TYPE Stream = [Int(INT32), eos];
PROCESS Generate(out : Stream; a, b, n : INT32);
VAR
    i : INT32;
BEGIN
    i := 0;
    WHILE i < n DO
        out ! Int(a + i*b); i := i + 1
    END ;
    out ! eos
END Generate;
```

An alphabet type `Stream` is defined first, containing two symbols. Symbol `Int` is associated with an `INT32` value, and symbol `eos` is a signal, carrying no data. The process contains a loop which outputs `n` integer values in the arithmetic progression over port `out`.

All examples from section 3.14.2 until section 3.14.4 assume that type `Stream` is defined as in this example.

3.14.2 Copy a bounded stream

This process copies a stream from an input channel to an output channel.

```
PROCESS Copy(inp, out : Stream);
VAR more : BOOLEAN; x : INT32;
BEGIN
    more := TRUE;
    WHILE more DO
        SELECT
            WHEN inp ? Int(x) THEN out ! Int(x)
            WHEN inp ? eos THEN more := FALSE
        END
    END ;
    out ! eos
END Copy;
```

Stellenbosch University http://scholar.sun.ac.za

This process contains a SELECT which is repeatedly executed until the end of the bounded stream is received. The SELECT can accept either an Int symbol or an eos signal from port inp. If an Int symbol is received, the integer value received is output to port out. If an eos signal is received, the loop terminates and the process outputs the eos signal to the output port.

3.14.3 Merge a bounded stream

If several guards of a SELECT are enabled, the command sequence associated with one guard is non-deterministically selected for execution. The process in the example accepts two input streams, and outputs an arbitrary interleaving of the two input streams.

```
PROCESS Merge(inp1, inp2, out : Stream);
VAR n, x : INT32;
BEGIN
    n := 0;
    WHILE n < 2 DO
        SELECT
            WHEN inp1 ? Int(x) THEN out ! Int(x)
            WHEN inp1 ? eos THEN n := n + 1
            WHEN inp2 ? Int(x) THEN out ! Int(x)
            WHEN inp2 ? eos THEN n := n + 1
        END
    END ;
    out ! eos
END Merge;
```

If Int symbols can be received from either inp1 or inp2, they will be received in any order and output on the out channel. Only when both input streams have terminated, will the loop terminate and the process will then send the eos signal over the output stream.

3.14.4 Suppress duplicates in a bounded stream

This example accepts an ordered input stream and outputs everything received except duplicate values.

```
PROCESS Suppress(inp, out : Stream);
VAR more : BOOLEAN; x, y : INT32;
BEGIN
    SELECT
        WHEN inp ? Int(x) THEN more := TRUE
        WHEN inp ? eos THEN more := FALSE
    END ;
    WHILE more DO
        SELECT
            WHEN inp ? Int(y) THEN
                IF x # y THEN out ! Int(x); x := y END
```

Stellenbosch University http://scholar.sun.ac.za


```
            WHEN inp ? eos THEN
                out ! Int(x); more := FALSE
        END
    END ;
    out ! eos
END Suppress;
```

A value is accepted and buffered until it can be compared with the next value. If the two consecutive values are different, the value received first is output and the second value is kept in the buffer. The first SELECT accepts the first value and keeps it for comparison. Provision is also made in the first SELECT for the eos signal, which ends the stream. The second SELECT accepts the following values and compares each value with the previously accepted value before outputting or discarding it. If the end of the stream is reached, the last received value is output and the eos signal is output before the process ends.

3.14.5 Library

LF implements modules, and allows modules to export and import processes, so a library can be written in the conventional fashion. Processes accessing global variables in a module can however not be exported; the compiler has no way of preventing processes in other modules from *calling* such a process simultaneously. Therefore interference of shared variables could have occurred if such processes were allowed to be exported.

Example:

```
MODULE ArrayUtils;

TYPE
    Arr* = ARRAY 100 OF INT32;
PROCESS ArrAvg*(VAR arr : Arr) : INT32;
VAR
    i, sum : INT32;
BEGIN
    i := 0; sum := 0;
    WHILE i < 100 DO
        sum := sum + arr[i];
        i := i + 1
    END ;
    RETURN sum DIV 100
END ArrAvg;

PROCESS ArrDiff*(VAR arr1, arr2 : Arr);
VAR
    i : INT32;
BEGIN
    i := 0;
    WHILE i < 100 DO
        arr1[i] := arr1[i] - arr2[i];
        i := i + 1
    END
```

Stellenbosch University http://scholar.sun.ac.za


```
END ArrDiff;

END ArrayUtils.
```

This library implements a set of operations on a predefined exported type `Arr`. The operations shown in the example are `ArrAvg` and `ArrDiff`. `ArrAvg` calculates the average value of the elements in an array and returns the answer as a result. `ArrDiff` calculates the difference between the corresponding elements of the two arrays and leaves the answers in the elements of the first array. The arrays are all passed to the processes as reference parameters, so the processes can only be *called.*

3.14.6 Abstract data structure

LF uses processes to encapsulate and abstract information. To implement an abstract data structure, the LF `SELECT` construct can be used. The synchronous communication serialises requests to the data structure, so the data in the structure is protected from concurrent access. In conventional languages, operations on abstract data structures are implemented by way of procedure calls. To mimic this, synchronous communication in LF is used to pass messages in which the symbol indicates the operation to be performed and the associated data the arguments, if any.

Example:

```
SELECT
WHEN list ? Update(key, field2, field3) THEN
    (* ... *)
WHEN list ? Delete(key) THEN
    (* ... *)
WHEN list ? NoOfEntries(entries) THEN
    (* ... *)
END ;
```

The "call" to the abstract data structure operation consists of

- the port (specifying which abstract data type to address),
- the symbol name (being an analogue of the procedure/method name) and
- the message data values (analogous to the actual parameters of a procedure call).

The `SELECT` not only serves to provide an interface to data for the rest of an implementation, it also serialises requests to that data. Therefore a `SELECT` provides a way of implementing a monitor.

Stellenbosch University http://scholar.sun.ac.za

3.14.7 Input/output

This example illustrates the use of a SELECT construct to serialise operations on a data structure and also to only accept requests meeting a specified criterion. The process Display contains an array of characters which has been placed with the AT constructs at the memory mapped to the text display. Any text written to the screen has to be processed by this process (unless another process in the system places variables in the screen memory).

Before any output requests can be accepted, the process wanting to output must obtain exclusive access to the screen by receiving a Claim message from the Display server. When receiving this message, the client process obtains a timestamp which must be supplied with each output request. Any process requesting output with an invalid timestamp will be blocked indeterminately. The process which has a valid timestamp can request to clear the screen or to write either a single character or a full display line to the screen. The client with the valid timestamp can also relinquish the screen when it has finished all output operations, by sending a Release message with a valid timestamp.

```
MODULE Screen;

CONST
    DisplayWidth = 80;
    DisplayHeight = 25;
    DisplayMemBase = 0B000H;
TYPE
    (* The display in memory contains 1 byte for every character,
        and 1 byte for the attributes (colour, brightness) of that
        character *)
    DisplayLineType = ARRAY DisplayWidth*2 OF CHAR;
    ScreenCommands* = [
        (* parm1 - timestamp of claim *)
        Claim(INT32),
        (* parm1 - timestamp of claim *)
        Release(INT32),
        (* parm1 - timestamp of claim; parm2 - text attribute *)
        ClearScreen(INT32, CHAR),
        (* parm1 - timestamp of claim; parm2 - line to write;
            parm3 - text attribute of line; parm4 - y coordinate *)
        WriteLine(INT32, DisplayLineType, CHAR, INT32),
        (* parm1 - timestamp of claim; parm2 - char to write;
            parm3 - text attribute of char; parm4 - x coordinate;
            parm5 - y coordinate *)
        WriteChar(INT32, CHAR, CHAR, INT32, INT32)];

(* port to accept commands on *)
PORT command* : ScreenCommands;
(* process managing text display *)
PROCESS Display(command : ScreenOutput);
TYPE
    TextScreenType = ARRAY DisplayHeight OF DisplayLineType;
VAR
    screenMem: TextScreenType AT DisplayMemBase;
    displayAttrib, ch : CHAR;
    line : DisplayLineType;
    x, y, curTime : INT32; (* position of next character on screen *)
```

Stellenbosch University http://scholar.sun.ac.za


```
    accessClaimed : BOOLEAN;

    (* clear display and set text attribute to specified type *)
    PROCESS ClearDisplayMem(attrib : CHAR);
    VAR
        i, j : INT32;
    BEGIN
        j := 0;
        WHILE j < DisplayHeight DO
            i := 0;
            WHILE i < DisplayWidth*2 DO
                screenMem[i][j] := 0X; screenMem[i + 1][j] := attrib;
                i := i + 2
            END ;
            j := j + 1
        END
    END ClearDisplayMem;

    (* check that given coordinates are within screen memory *)
    PROCESS RangeCheck(x, y : INT32):BOOLEAN;
    BEGIN
        IF (x >= 0) & (x < DisplayWidth) &
            (y >= 0) & (y < DisplayHeight) THEN
            RETURN TRUE
        END ;
    RETURN FALSE
    END RangeCheck;

BEGIN
    ClearDisplayMem(screenMem, DefaultCharAttrib);
    WHILE TRUE DO
        (* grant exclusive access to a process *)
        TIME ? NOW(claimTime);
        command ! Claim(claimTime);
        accessClaimed := TRUE;
        (* accept requests from process with exclusive access *)
        WHILE accessClaimed DO
            SELECT
                WHEN command ? ClearScreen(time, displayAttrib) & time = claimTime THEN
                    ClearDisplayMem(displayAttrib)
                WHEN command ? WriteLine(time, line, displayAttrib, y) & time = claimTime THEN
                    IF (y >= 0) & (y < DisplayHeight) THEN
                        x := 0;
                        WHILE x < DisplayWidth DO
                            screenMem[x*2][y] := line[x];
                            screenMem[x*2 + 1][y] := displayAttrib;
                            x := x + 1
                        END
                    END
                WHEN command ? WriteChar(time, ch, displayAttrib, x, y) & time = claimTime THEN
                    IF RangeCheck(x, y) THEN
                        screenMem[x*2][y] := ch;
                        screenMem[x*2 + 1][y] := displayAttrib
                    END
                WHEN command ? Release(time) & time = claimTime THEN
                    accessClaimed := FALSE
            END
        END
    END
END Display;

BEGIN
    NEW(command);
    CREATE Display(command)
END Screen;
```

Stellenbosch University http://scholar.sun.ac.za

3.15 Modifications to the language for this thesis

In Section 3.1 the previous version of the LF language was discussed, and several shortcomings were highlighted. Below, areas are listed where the new version of LF improves upon the previous version. For each area at least one example of an improvement in the language is given.

3.15.1 Limiting user freedom to promote safety

Pointers have been removed from the language, since pointers can be used to implement shared memory for processes (see Section 3.4.1), and misuse of pointers can hinder model checking efforts (see Section 3.1).

3.15.2 Improved control over concurrency

The previous version of LF only implemented process creation where the newly created process runs concurrently with its parent. In the new version, processes can be *called*, or instantiated so that the caller is blocked until the callee terminates (see Section 3.6). This provides procedure call semantics in the language.

Because of this feature, some processes can be shown at compile time never to execute concurrently, and data can be shared between such processes. The new version of LF therefore supports sharing of data and passing of reference parameters between caller and callee (see Section 3.12). This will lessen the need for copying in implementations.

3.15.3 Abstraction

The language mechanism of *calling* a process discussed in Section 3.15.2 provides a way to implement subroutines — still one of the most widely used abstraction tools in modern programming languages (see Section 3.6.1).

The `MODULE` provides a way to modularise implementations and make code reuse possible (see Section 3.9). Because the memory of modules (and therefore the channels belonging to modules) stay allocated while the system is running, channels can be exported to form interfaces between hidden processes and the rest of the system.

Stellenbosch University http://scholar.sun.ac.za

3.15.4 Constructs to support hardware-level programming

The `SYSTEM` module provides an interface between LF processes and the hardware on which the LF system executes (see Section 3.11). From this `SYSTEM` module interrupt channels are exported, to be used by device driver processes executing in user space.

The `TIME` channel provides a way for processes to measure the passing of time.

3.16 Summary

In this chapter the language LF was presented. This thesis is a continuation of work started in [21]. A simple example of the original version of this language was given in section 3.1, and several shortfalls of that prototype were discussed. The language is intended for faster development of reliable embedded systems. Correctness is the most important goal, although efficiency of developed systems is also important. To achieve this, several design goals were set in section 3.2. These included, among others, suitability for a verification technique known as model checking, encouragement of safe programming practices, and low-level operations for interfacing with hardware.

The language is CSP-based, similar to occam and Joyce. Therefore an LF system consists of processes communicating via synchronous message passing. Processes can execute concurrently, or can be instantiated to run to completion. All forms of data sharing between concurrent processes, including pointers, are disallowed. Instead, such processes can only be transmitting messages which are passed between processes by channels. The messages, and all other data, are subject to strong type checking to detect errors at compile time.

The syntax of the language is based on the syntax of the Oberon language. The module concept from Oberon was also used to provide modularity to the language. Abstraction tools such as modularity and abstract data structures provide ways to divide a complex implementation into sections which can be model checked separately.

Low-level constructs to interface with hardware were discussed. These include interrupt channels, placing variables at specific absolute addresses in memory and a tool to measure the passing of time. Scoping rules are used to ensure that concurrent processes do not share data. It is claimed that the language disallows processes to interfere in and corrupt the data of other concurrent processes, and finally, some examples of LF code is given.

Stellenbosch University http://scholar.sun.ac.za

Chapter 4

Code Generation to Support Model Checking

It is of course possible to generate machine code from LF source code in the traditional way. However, since the goal is to support model checking of LF programs, machine code is generated in the form of a transition system and executed by an efficient interpreter. The major advantage of this approach is that the number of interleavings between different execution paths is reduced considerably. This is so because rescheduling can only occur at predetermined points in the machine code, and not after every machine instruction.

According to [1, Chapter 2], a *transition system* is a quadruple

$$\mathcal{A} =< S, T, \alpha, \beta > \textit{ where}$$

- S is a finite or infinite set of *states*
- T is a finite or infinite set of *transitions*
- α and β are two mappings from T to S which take each transition t in T to the two states $\alpha(t)$ and $\beta(t)$, respectively the *source* and the *target* of the transition t.

In other words, a transition modifies the current state of the transition system. In the transition system generated from LF source code, S and T are both finite sets. When a transition between two states occurs, an an atomic *action* that usually consists of several machine instructions is executed. Such an action may update the data variables of the system, and thus transform the system from one state to another. Actions are viewed as atomic

Stellenbosch University http://scholar.sun.ac.za

even though code is executed with interrupts enabled, because the system only responds to interrupts after completion of each action. Rescheduling, which takes place after servicing of interrupts, can only happen between transitions. Each different scheduling point increases the ways in which the scheduler can order process code, so reducing the scheduling points will reduce the state space for model checking. The details of code generation and exactly what is meant by execution of transitions are discussed in what follows.

4.1 Goals

- The primary goal is to generate executable code structured as actions or transitions.
- Because servicing of interrupts are delayed, the execution time of actions have to be limited, so the system does not lose interrupts. However, too short actions are also not desirable, since these will increase the overhead of the transition system. Because control is transferred to and from actions by an interpreter, smaller actions will incur more interpreter overhead when executing the system. Therefore, for best results a balance has to be found between responsiveness (actions short enough) and efficiency (actions long enough).

4.1.1 The Intel 386 processor and assembly language

Code examples in this chapter are written in the in-line assembly language for the Intel 386 processor as supported by the ETH Oberon compiler. The 80386 is widely used for embedded software. Many different assembler languages exist, so a short introduction of the format used in this document is given.

The Intel 386 architecture has eight 32-bit general-purpose registers: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `ESP` and `EBP`. The lower 16 bits of all these registers can also be used as 16-bit registers, and four of these 16-bit registers are further subdivided into 8-bit registers. According to Intel convention an 8-bit value is called a *byte*, a 16-bit value a *word* and a 32-bit value a *doubleword*.

Intel assembly language follows a general format for specifying an instruction:

```
<opcode> <destination operand> <source operand>
```

If an operand is a number, it specifies a constant; a string can specify one of the registers, or

Stellenbosch University http://scholar.sun.ac.za

a variable, constant or label. Square brackets indicate indirect addressing; inside the square brackets an expression can be specified to serve as indirect address. The opcode can be preceded by a label.

Some examples:

- Transfer execution unconditionally to a location specified by label `UNTIL`.

```
JMP UNTIL
```

- Pop a value from the stack into the 32-bit register `EAX` (the value popped will also be a 32-bit value).

```
POP EAX
```

- Compare the value at the address specified by the first operand to 0 and set the condition flags accordingly. The first operand is determined by interpreting the value in register `EDI` as a base address and adding the constant `symFirstReader` as offset. The size of the operand can be specified with the `DWORD` specifier, as in the following example.

```
CMP DWORD symFirstReader[EDI], 0
```

- Jump to a location specified by label `AlreadyQueued`, if "not equal" was the result of the last expression evaluated.

```
JNE AlreadyQueued
```

Details regarding instruction mnemonics and instruction formats for the Intel IA-32 architecture, which includes the instructions for the 80386, can be found in [5].

4.2 Transitions generated by the LF compiler

Figure 2 illustrates a single transition forming part of the LF transition system. A quick discussion of how to interpret the Figures follows. A discussion on the generation of machine code from LF commands is done separately, in Section 4.3.2.

The action in Figure 2 consists of three assignments and a jump to the interpreter. The leftmost column shows the absolute address where the code will be loaded in memory. This is

Stellenbosch University http://scholar.sun.ac.za

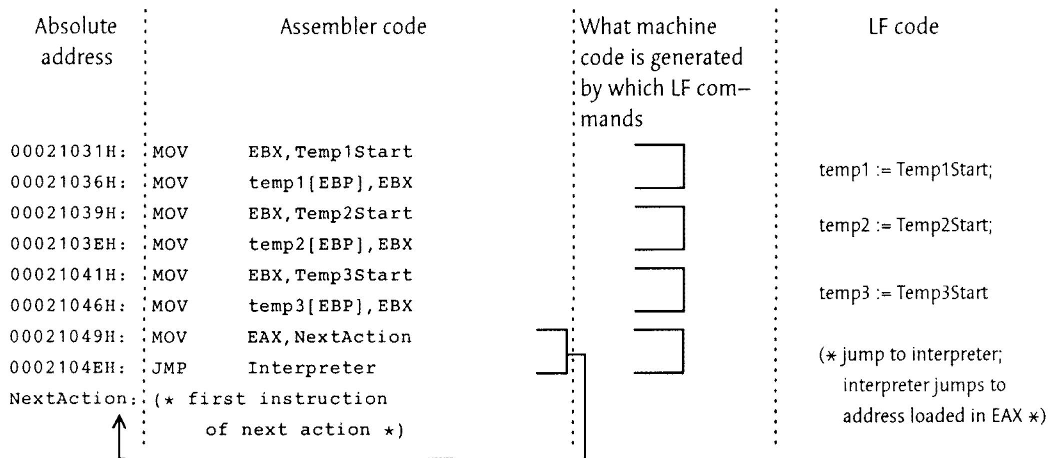

Figure 2: A simple action, consisting of three assignments

followed by the assembler code representing the machine instruction. The rightmost column shows the LF code which generated the machine code, and the column in between shows which LF commands yielded which machine instructions. At address `21031H`, the constant `Temp1Start` is loaded into register EBX. The next instruction stores the value in EBX to a memory location, calculated as follows: register EBP contains the base reference to the activation record of the current process, and `temp1` is the offset of variable temp1 in the activation record. The next two assignments are similar to the first.

Every action ends with a jump to the interpreter, as shown in the last two lines. At address `21049H` the address of the next action is loaded into a register for use by the interpreter. All other registers are assumed not in use, so no registers will be saved by the interpreter. Next, the action transfers control to the interpreter with a jump instruction.

4.2.1 The interpreter

Pseudocode for the interpreter is listed in Figure 3. It first examines whether any interrupts have occurred during the last action (line 0). If so, interrupts are disabled (line 1), and an IPC message is sent to the appropriate interrupt handler process for every interrupt which has occurred (line 2–12).

Stellenbosch University http://scholar.sun.ac.za


```
0   BEGIN
1     IF Interrupts # {} THEN
2       DisableInterrupts;
3       REPEAT
4         FindNext(Interrupts, InterruptNo);
5         EXCL(Interrupts, InterruptNo);
6         InterruptChannel := DeviceChannel[InterruptNo];
7         IF WaitingToReceiveOn(InterruptChannel) THEN
8           SendMessage(InterruptChannel);
9           UnblockReceiver(InterruptChannel)
10        ELSE
11          QueueMessage(InterruptChannel)
12        END;
13      UNTIL Interrupts = {};
14      EnableInterrupts;
15      Reschedule() (* NOTE: exit point from procedure *)
16    END;
17  Continue() (* execute next action *)
18  END Interpreter;
```

Figure 3: Pseudocode for the interpreter

While actions are executing, interrupts are enabled. Simple interrupt handler procedures execute directly after each interrupt. They mark interrupts as having occurred, without servicing them. The interrupt is not acknowledged on the device itself — actions have to be short enough so that an interrupt on any device is serviced before the next interrupt occurs on that device. A set of bits is maintained (called `Interrupts` in the pseudocode) and each simple interrupt handler sets a different bit here and acknowledges the interrupt on the interrupt controller. When the interpreter executes (after every action), the set of bits is examined (line 0). If a bit is marked, the interpreter clears the bit (line 4) and converts the interrupt to an interprocess communication message. If an interrupt handler process is waiting to receive a message over the interrupt channel, the message is sent to that interrupt handler process (not part of the kernel), which services the interrupt, and acknowledges the interrupt on the device. If no process is waiting to receive over the channel, the interrupt is queued.

If interrupts have occurred during the previous action, interrupts (disabled on line 1) are enabled on line 13. Then the scheduler is called (line 14). The scheduler may change the currently running process before the next action begins, and does not return to the interpreter, but transfers control to the next action of the new current process itself. If no interrupts have

Stellenbosch University http://scholar.sun.ac.za

occurred, the interpreter transfers control to the next action of the currently running process (line 16).

4.3 Implementation of the compiler

The design of the compiler is based on the Oberon-0 compiler as described in [24], and the OP2 (Oberon) compiler, as discussed in [6]. Since LF is an experimental language, emphasis was placed on simplicity and maintainability rather than optimisation or compilation speed in the design of the compiler. The aim was to get an initial version of the compiler working that is simple and reliable. Maintainability is also enhanced if the compiler is similar to other well-documented or well-known compilers. This is one reason why the examples of the Oberon compilers were followed in many areas.

A recursive descent parser is used and an intermediate form of the program is generated in the form of a syntax graph. The format of this graph is based upon the format of the graph generated by the OP2 compiler, as described in [6].

4.3.1 Symbol table

The symbol table used in the LF compiler is based on that of the Oberon-0 compiler. To illustrate the layout of the symbol table, a declaration sequence is given below, and the corresponding layout of the symbol table is shown in Figure 4. These show the layout of the symbol table for variables of composite (array and record) types.

```
TYPE
   RecordType = RECORD
      intField1 : INT32;
      charField1, charField2 : CHAR
   END ;
   ArrayType = ARRAY 10 OF CHAR;
VAR
   rec1 : RecordType;
   arr1, arr2 : ArrayType;
   int : INT32;
```

The symbol table is organised as a stack of linked lists. This is so because LF is a language with Oberon-like local scopes for procedures. Each scope in the symbol table will have relatively few entries since each process defines only the variables constants and types it needs. Therefore intricate data structures to optimise searches would yield no substantial benefit over a list,

Stellenbosch University http://scholar.sun.ac.za

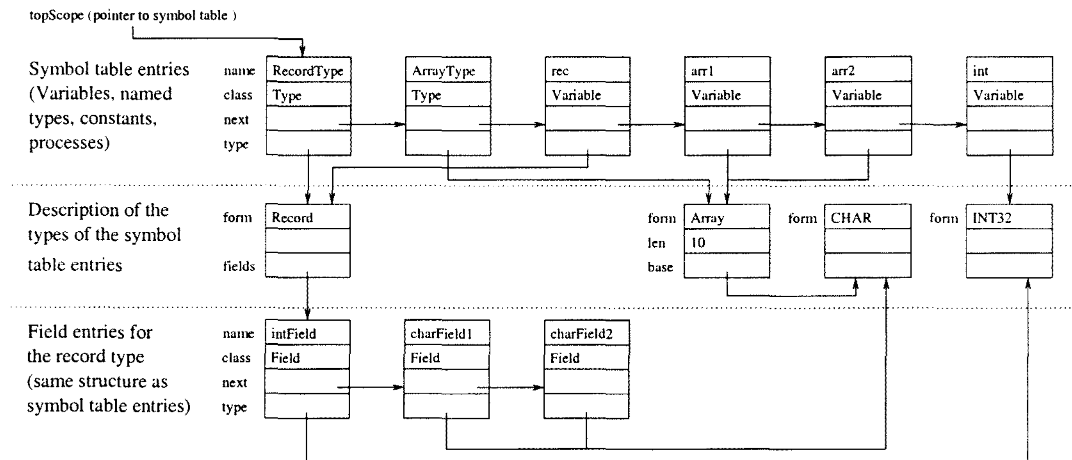

Figure 4: Layout of the symbol table for composite data types

yet require more overhead to maintain, and be more complicated to implement. This is also the situation in Oberon [24, Chapter 8].

To illustrate the structure of the symbol table, an LF module is given below, and the layout of the symbol table for this module is shown in Figure 5.

```
MODULE MyFirst;
VAR
    x, y, INT32;

    PROCESS Container(x : CHAR);
    VAR
        y : INT32;
        PROCESS Contained(y : CHAR);
        VAR
            a, b : INT32;
        BEGIN
            y := x (* symbol table at this point shown *)
        END Contained;
    BEGIN
        IF x = "h" THEN y := 35 END
    END Container;
BEGIN
    Container("h")
END MyFirst.
```

The entities `INT32` and `CHAR` are symbol table entries with `type` fields pointing to the appropriate type description records; the entries are only there so the names of these named types can be stored in the symbol table. For the sake of simplicity the type descriptions have been omitted from the diagram.

The current scope is the top level of the stack, with all encapsulating scopes beneath it.

- The lowest scope contains all entries for standard types and predefined entities such as

Stellenbosch University http://scholar.sun.ac.za

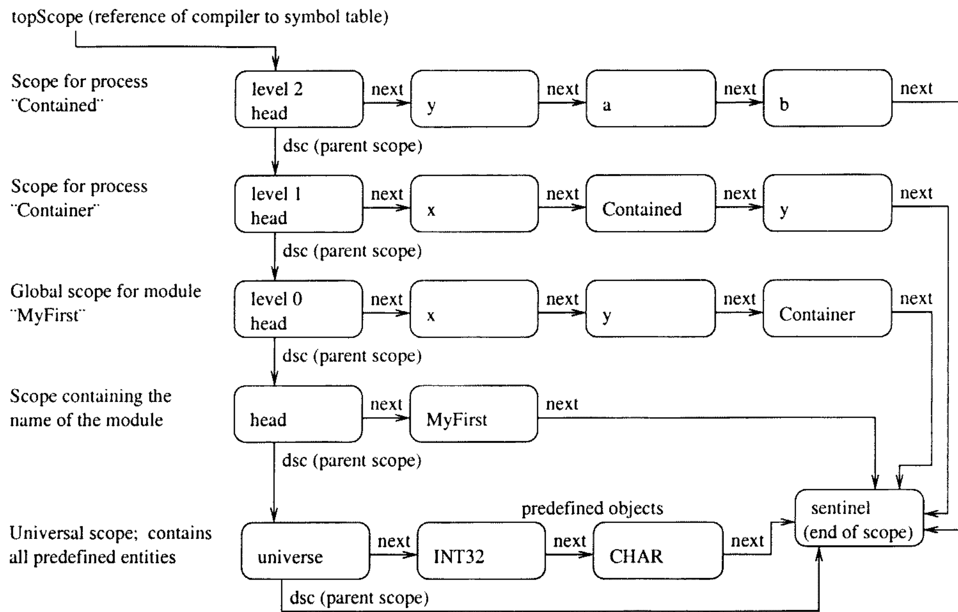

Figure 5: Layout of the symbol table

TRUE and FALSE.

- In the scope just above the lowest scope is an entry for the module.
- The scope directly above the module scope contains all global entities such as global ports, global constants and global processes.
- All scopes above that contain entities defined in nested processes.
- The end of each scope is indicated by the `sentinel` entry.

A symbol is located by searching left to right, top to bottom; if a symbol is not found in the current scope, the search is continued in the parent scope. The next symbol in the current scope is found by dereferencing the `next` pointer, and the parent scope is found by following the `dsc` pointer.

Refinements and problems

Even though the syntax of LF is similar to that of Oberon, the LF compiler needs to implement functionality not needed in an Oberon compiler.

First, an LF process is currently limited to only be *called* (not *created*) once it references variables declared in a parent scope (see Section 3.12). The compiler enforces this by marking

Stellenbosch University http://scholar.sun.ac.za

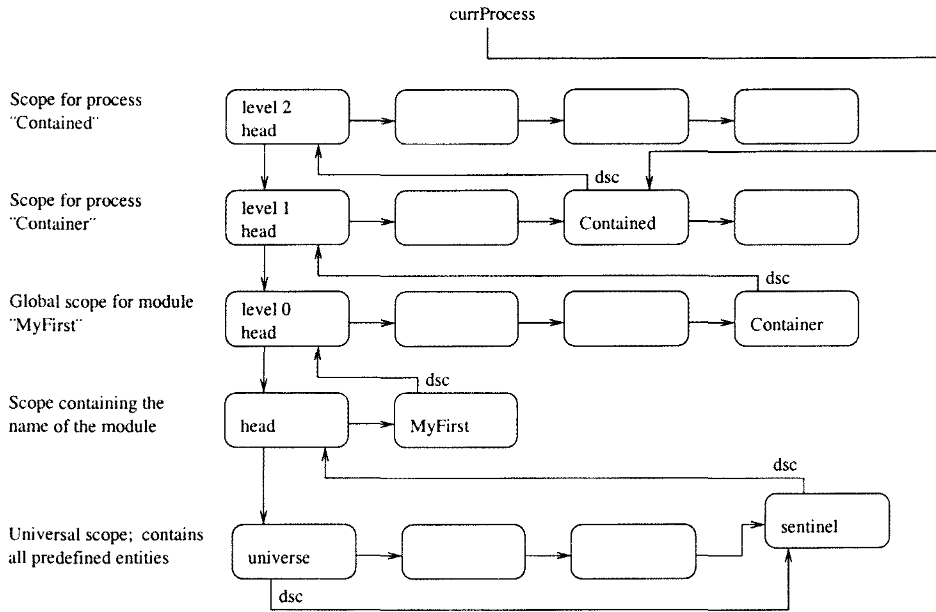

Figure 6: The pointer to the current process in the symbol table. This sample symbol table corresponds to the same code as Figure 5.

the current process as *call*-only once it references a variable declared outside the top (current) scope. A pointer to the current process is maintained for these purposes, as can be seen in Figure 6. The pointer is maintained in the following fashion:

- While process *Contained* is being compiled inside encapsulating process *Container*, the `dsc` pointer of entry `Container` in the symbol table points to the scope of process *Contained*, and the `currProcess` pointer points to entry `Contained`.
- When the compiler has finished compiling process *Contained*, entry `Container` is located (its `dsc` pointer points to the current scope) and pointer `currProcess` is pointed to this entry.

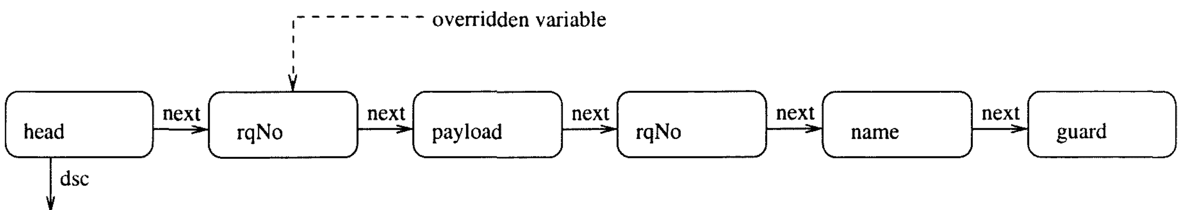

Figure 7: The top scope of the symbol table showing overriding of variables. This is done during compilation of the boolean expression in a receive guard for a `SELECT` command.

The second modification to the symbol table was done to provide for a specific LF construct. A problem arose when implementing the code generation for a receive guard in the `SELECT`

Stellenbosch University http://scholar.sun.ac.za

construct. The boolean condition in a receive guard may contain variables which might change if the message is received. However, inside the boolean expression these variables have to temporarily assume their new values, even though the message has *not* been received yet (see Section 2.3.3 and Section 3.8).

To do this, the variables temporarily address the values to be transmitted *inside the memory of the sender process.* This is possible because all LF processes on a system run in one address space; memory protection is implemented by the compiler. Of course the values of expressions, which are always calculated in registers, need to be saved to memory to allow this.

In the example below, `rqNo` is such a variable. It will be received if the guard evaluates to TRUE, in other words if:

- the communication is found to be feasible and
- the condition `rqNo = bufferNo` is TRUE.

```
SELECT
    WHEN op ? PrintRequest(name, rqNo, payload) &
    rqNo = bufferNo THEN
        (* accept new print job *)
        InsertEntry(head, name, payload);
        head := (head + 1) MOD BufferSize
```

To temporarily *override* the normal addressing of the variables, other entries with the same names but different addresses are inserted earlier in the symbol table. It would have been convenient to just open a new nested scope and put all temporary entries into this new scope. However, suppose other variables, not overridden, were referenced inside the boolean expression (like variable `bufferNo` in the example). If overriding variables were put in a new scope, it would have "seemed" to the compiler that non-local variables (outside the top scope) were referenced. The compiler would have then restricted the current process to only be *called*. To overcome this problem, it was decided to insert new entries for the *overridden* variables at the front of the same scope, and remove them after compiling the boolean expression. Shown in Figure 7 is the top scope of the symbol table. It illustrates variable overriding during compilation of the boolean expression in the *Receive* guard of the SELECT example above.

Stellenbosch University http://scholar.sun.ac.za

4.3.2 Tree structure and code generation

The syntax graph used to represent the code has as basis a conventional binary tree. Therefore each node has a `left` and `right` field, pointing to the left and right subtrees. In addition, each node has a `link` field, used to point to the next command in a block of commands. The rationale for the `link` field is that structures like parameter lists and command sequences occur often in LF. Maintaining a list via `link` fields is much more efficient than inserting nested subtrees with unused nodes.

During code generation the syntax tree is traversed in a recursive fashion. A jump to the interpreter (which ends an action) is normally inserted after the machine code for a command. However, several assignment commands can be composed into one action, as will be shown below.

Assignments

To illustrate code generation for assignments, an example of three consecutive assignments is given below. The assignments initialise three variables. The syntax graph for the three assignments is shown in Figure 8.

```
CONST
    Temp1Start = 0;
    Temp2Start = 15;
    Temp3Start = 30;
VAR
    temp1, temp2, temp3 : INT32;
BEGIN
    temp1 := Temp1Start;
    temp2 := Temp2Start;
    temp3 := temp1;
    (*...*)
```

Corresponding machine code for these LF commands are shown in Figure 9. Local variables of the process are addressed relative to register `EBP`. `EBP` contains the base address of the local data area belonging to the current process; this data area is called the *activation record* of the process. The terminology ("activation record") and use of a base pointer register are reminiscent of procedure activation frames. However, memory for processes is not allocated according to the stack principle. Rather, the run-time system allocates memory for a process, and maintains a reference to the base of this memory area in register `EBP` when the process executes.

Stellenbosch University http://scholar.sun.ac.za

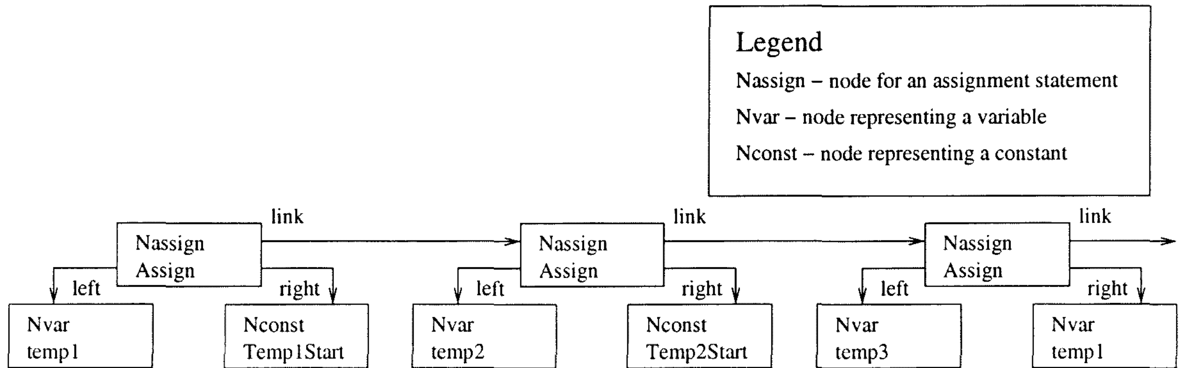

Figure 8: Intermediate format for three consecutive assignments

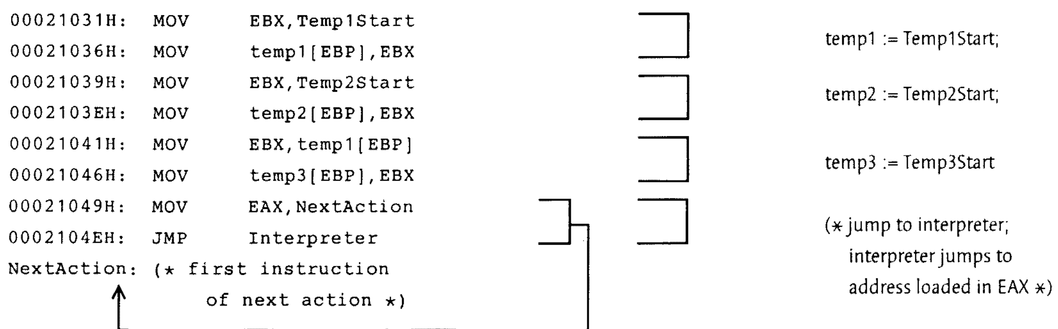

Figure 9: Intel 386 code for three consecutive assignments; one action

The three assignments are compiled to a single action. In the first two commands (addresses `21031H` to `2103EH`) a constant is assigned to a variable, and in the third command (addresses `21041H` to `21046H`) a variable is assigned to another variable. Addresses `21049H` and `2104EH` contains the end of the action. After each action the start address of the next action is loaded into register `EAX`, and control is transferred to the interpreter.

As mentioned in Section 4.1, actions cannot take too long to execute, or interrupts will be lost. To limit the duration of actions, the compiler currently counts the number of instructions, and inserts a jump to the interpreter if this count becomes too large. Ideally the compiler should count the number of clock cycles an action needs to execute, and not the number of instructions. However, to do this the compiler needs an indication of the worst-case number of clock cycles each instruction would need to execute. Creating a database with such estimates was considered too much effort with too little value for the experiment discussed in this thesis.

Stellenbosch University http://scholar.sun.ac.za

Control structures

As first example of a control structure, the code generation for an IF command is shown below. An IF with one ELSIF clause and one ELSE clause is shown.

```
IF (x1 = ValidX1) THEN
    x2 := ValidX2; x3 := ValidX3 (* command sequence 1 *)
ELSIF (x2 = ValidX2) THEN
    x1 := ValidX1; x3 := ValidX3 (* command sequence 2 *)
ELSE
    x1 := ValidX1; x2 := ValidX2; (* command sequence 3*)
    x3 := ValidX3
END ;
```

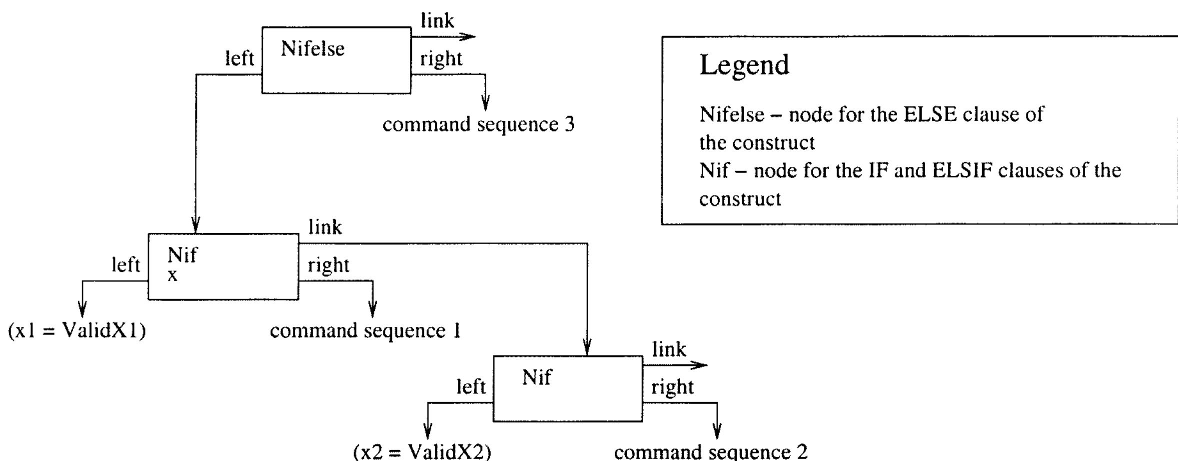

Figure 10: Syntax tree for a sample IF..ELSIF..ELSE command

The syntax graph for an IF command (shown in Figure 10) is arranged so that the *Nifelse* node, representing the ELSE clause, forms the root of the tree, with its link field pointing to the next command after the IF..ELSIF..ELSE. The left subtree of the *Nifelse* contains a list of all the subtrees for the IF and ELSIF clauses, while the right subtree contains all the commands in the ELSE clause. For each *Nif* node, representing an IF or ELSIF, the left subtree contains the boolean condition, and the right subtree the commands of that clause.

The code for the command is generated in such a fashion that short IF..ELSIF..ELSE commands can be executed as one action. Figure 11 illustrates how code is composed into actions for an IF..ELSIF..ELSE command. The code is arranged so that a jump to the interpreter is executed only after a IF, ELSIF or ELSE clause has been executed. For example, if the condition for the ELSIF clause (Figure 11, address 21058H) evaluates to TRUE, the action executed will consist of the expressions x1 = ValidX1 and x2 = ValidX2, and the command sequence labelled command sequence 2. If an ELSE clause is not present,

Stellenbosch University http://scholar.sun.ac.za

a jump to the interpreter will be executed if all expressions have been evaluated and all were found FALSE.

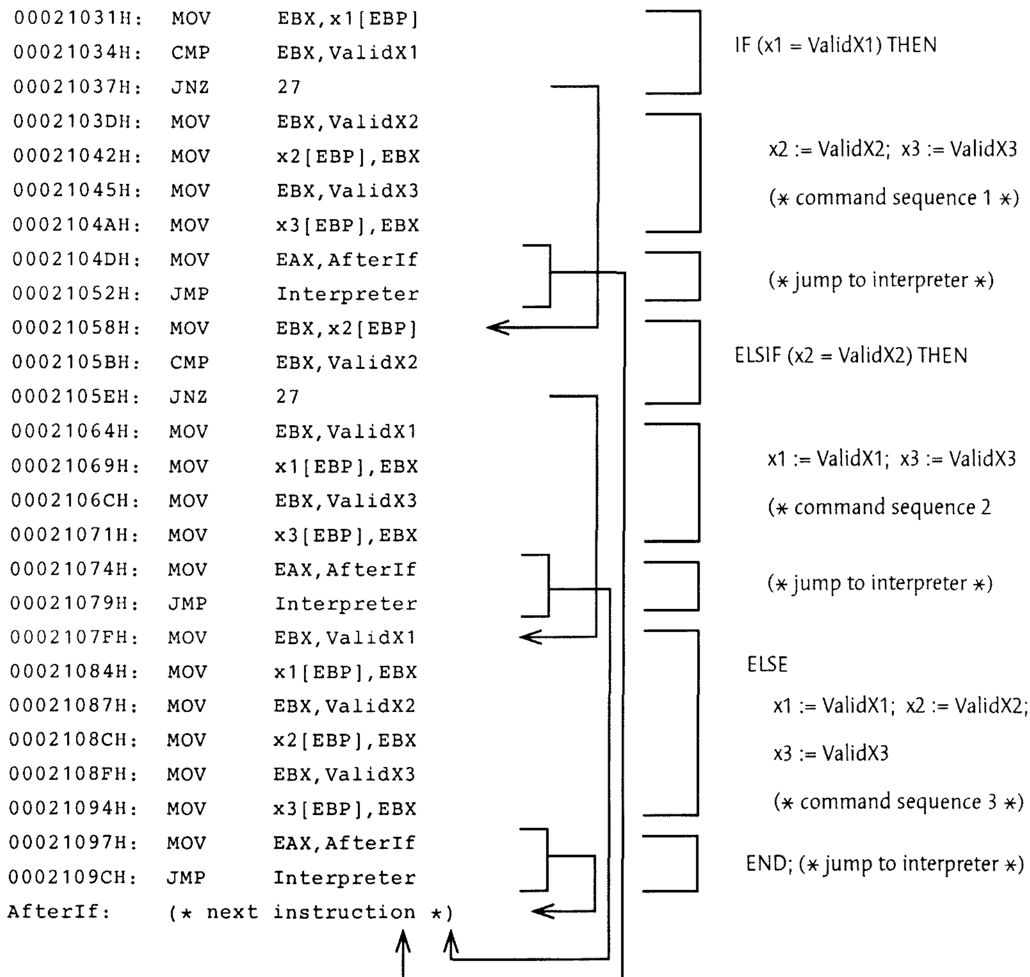

Figure 11: Intel 386 code generated for a sample IF..ELSIF..ELSE command

Code is generated in a similar fashion for the WHILE command. The LF code for the example is given below, and the corresponding syntax tree is shown in Figure 12.

```
WHILE (x1 = ValidX1) & (x2 = ValidX2) OR (X3 = ValidX3) DO
    x1 := ValidX1
END ;
```

A more complex boolean condition was used for the WHILE example than for the IF to illustrate the short-circuit evaluation of expressions. Short-circuit evaluation entails not evaluating parts of the boolean expression which will not change the value of the expression. This

Stellenbosch University http://scholar.sun.ac.za

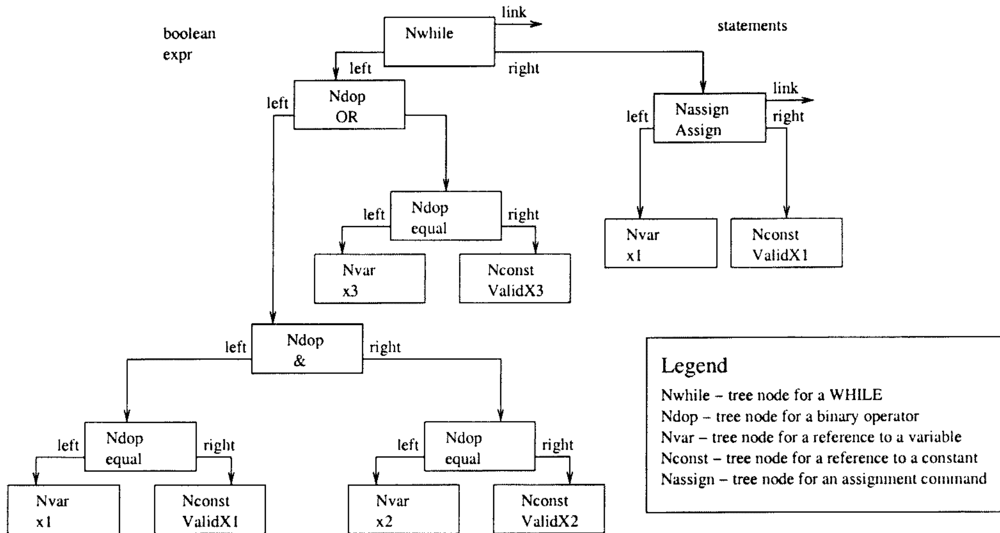

Figure 12: Syntax tree for a sample `WHILE` command

is implemented by inserting conditional jumps into the code for the expression, as illustrated in Figure 13.

Because a `WHILE` command is not guaranteed to terminate, it cannot be executed in only one action. Rather, a `WHILE` command with a body consisting of one action is divided into one action per iteration. As illustrated in Figure 13, the body of the `WHILE` is ended with a jump to the interpreter (address `2105DH` and `21062H`). The interpreter will return control to the beginning of the `WHILE` when execution of the process resumes. Therefore the decision to terminate the loop or not will only be made when the process resumes with the next action.

Interprocess communication

Communication between processes is implemented by the run-time system. Therefore, a process needing to send data to another process needs to make a system call. This call will be generated by the compiler, from the information available from the *Send* or *Receive* command. An example of a *Send* command is given below.

```
TYPE
    ChanType = [a(INT32), b(INT32, INT32)];
VAR
    x1, x2, x3 : INT32;
    temp1, temp2, temp3 : INT32;
PORT
    port, port2 : ChanType;
BEGIN
    port ! a(x1);
    (*...*)
```

Stellenbosch University http://scholar.sun.ac.za

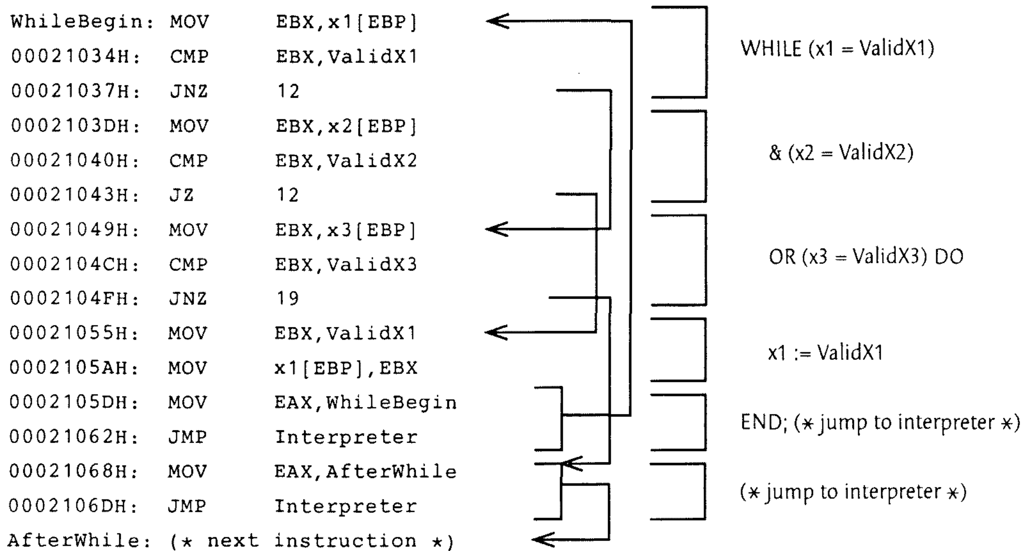

Figure 13: Intel 386 code generated for a sample `WHILE` command

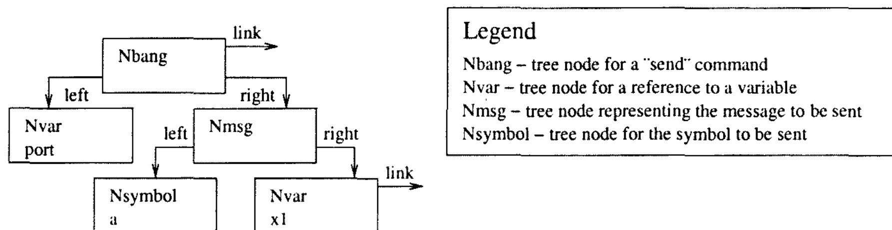

Figure 14: Syntax tree for a sample *Send* command

The code generated to make the system call is given in Figure 15. Information is passed to the runtime system by loading it in specific registers before the system call, and the system call is a jump to a destination where the code for the system call starts. The following happens in Figure 15:

- line 0: A reference to the variable to be sent is loaded. The run-time system expects it in register `EDI`.
- line 1: The port (referring to the channel over which the message will be sent) is loaded into register `EBX`.
- line 2: The size of the data to be copied is loaded into register `ECX`.
- line 3: The system call needs to know which symbol is being sent. The data structure for a channel contains a queue for every symbol in the channel alphabet (see Section 3.7.2

Stellenbosch University http://scholar.sun.ac.za


```
0.  00021037H:  LEA   EDI,x1[EBP]                  reference to variable x1
1.  0002103AH:  MOV   EBX,port[EBP]                reference to channel (stored in port)
2.  0002103DH:  MOV   ECX,Size(x1)                 message size
3.  00021042H:  LEA   EDX,12[EBX]                  address of symbol in channel record
4.  00021045H:  MOV   EAX,NextAction               address of next action
5.  0002104AH:  JMP   SendMessageSystemCall        system call to send message
    NextAction:
```

Figure 15: Intel 386 code generated for a sample *Send* command

and Figure 1). The address of this queue is loaded into register EDX.

- line 4: The system call is also the end of an action. Therefore the address of the next action (of this process) after the system call is loaded into register EAX.
- line 5: The system call is made by jumping to the address where the code for the system call starts.

The SELECT construct

The SELECT construct increases the expressiveness of the language substantially, because several communication options can be given of which one should be executed. The command also implements a *conditional receive*, as discussed in Section 3.8, allowing a process to examine a message *before* receiving it. However, implementing the command efficiently is challenging, as will be illustrated.

A simple example of a SELECT command is given below — it consists of one output guard attempting to send the signal SymSignal, and one input guard attempting to conditionally receive the SymInt symbol. The syntax graph and code generated for each WHEN clause is discussed separately afterwards.

```
SELECT
TYPE SimpleChanType = [SymSignal, SymInt(INT32)];

PROCESS Server;
VAR x, y : INT32;
PORT p : SimpleChanType;
BEGIN
    (* ... *)
    SELECT
        WHEN p ! SymSignal THEN y := 1
        WHEN p ? SymInt(x) & x > 5 THEN y := 2
    END
END Server;
```

Stellenbosch University http://scholar.sun.ac.za

In Figure 16 the tree structure for a SELECT subtree is shown. The SELECT subtree follows the same convention as that of the IF..ELSIF..ELSE subtree. However, since the SELECT construct has no ELSE clause, the top right subtree of a SELECT tree will always be empty.

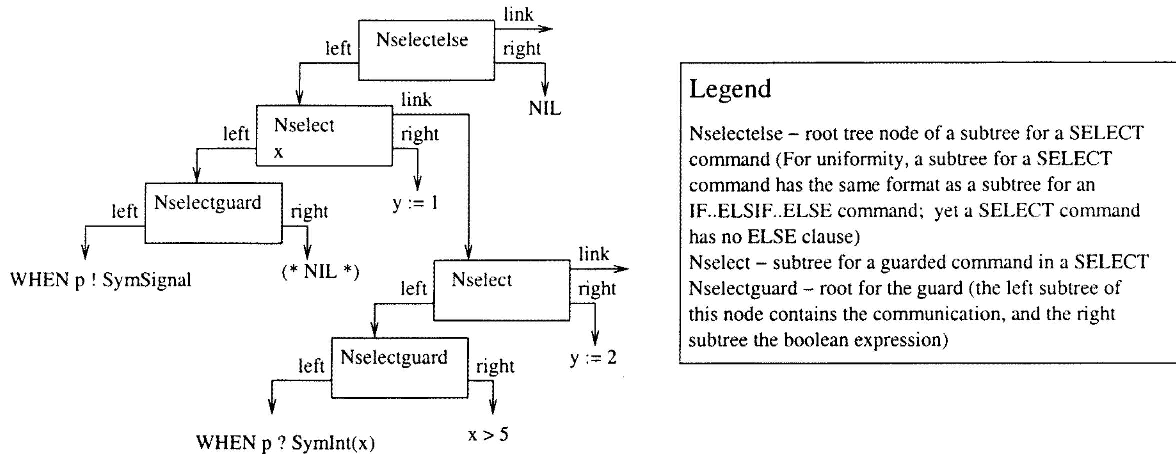

Figure 16: Syntax tree for the complete SELECT command

The logic of the executable code produced for a SELECT command is illustrated below in the form of pseudocode. If a WHEN guard is not executable, the next action to be executed in this process will be the next WHEN guard. If no guard is executable, a Reschedule system call will be executed. Other processes will now be scheduled, some of which might enable some of the WHEN guards. When this process is scheduled again, execution will start once more at the first guard.

```
LOOP
   IF Executable(p ! SymSignal) THEN
       Execute(y := 1); Exit
   ELSIF Executable(p ? SymInt(x)) & x > 5 THEN
       Execute(y := 2); Exit
   ELSE Reschedule
   END
END
```

With the logic of the executable code for a SELECT explained, we can examine the machine code for the guards themselves in more detail. An illustration of the executable code for the *Send* guard is given in Figure 17.

The *Send* guard contains a system call similar to the system call for the *Send* command discussed in Section 4.3.2. The following information is loaded into specific registers to transfer it to the system call:

- line 0: The value of the expression to be sent (for a signal this will be a dummy value)

Stellenbosch University http://scholar.sun.ac.za

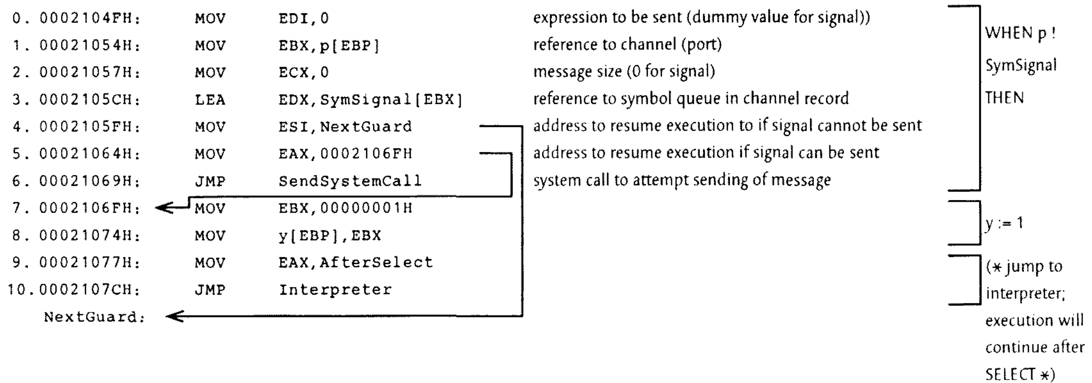

Figure 17: Intel 386 code generated for the first `WHEN` clause (containing a *Send* guard) of the first `SELECT` example

is loaded into register `EDI`.

- line 1: A reference to the channel over which the message will be sent is loaded into register `EBX`.
- line 2: The message size (0 for a signal) is loaded into `ECX`.
- line 3: A reference to the queue for this symbol (part of the data structure representing the channel) is loaded into `EDX`.
- line 4: The address of the action to execute if the signal cannot be sent (containing the next guard in the `SELECT`) is loaded into `ESI`.
- line 5: The address of the action to execute if the signal can be sent (`y := 1`) is loaded into `EAX`.

If the signal can be sent, the command `y := 1` will be executed next. After a jump to the interpreter, execution will continue after the `SELECT` command.

If the signal *cannot* be sent, the next guard will be tested by executing the code listed in Figure 18. A *Receive* guard may contain a boolean expression using values which have not been received yet. Therefore a *Receive* guard needs two system calls. The first system call examines whether communication is possible. If it is, the run-time system supplies the address of the data to be received in register `ESI`, so the boolean expression can reference the data. If the boolean expression evaluates to `TRUE`, the second system call is executed to indicate to the runtime system to copy the message. The code for the first system call is discussed below.

Stellenbosch University http://scholar.sun.ac.za

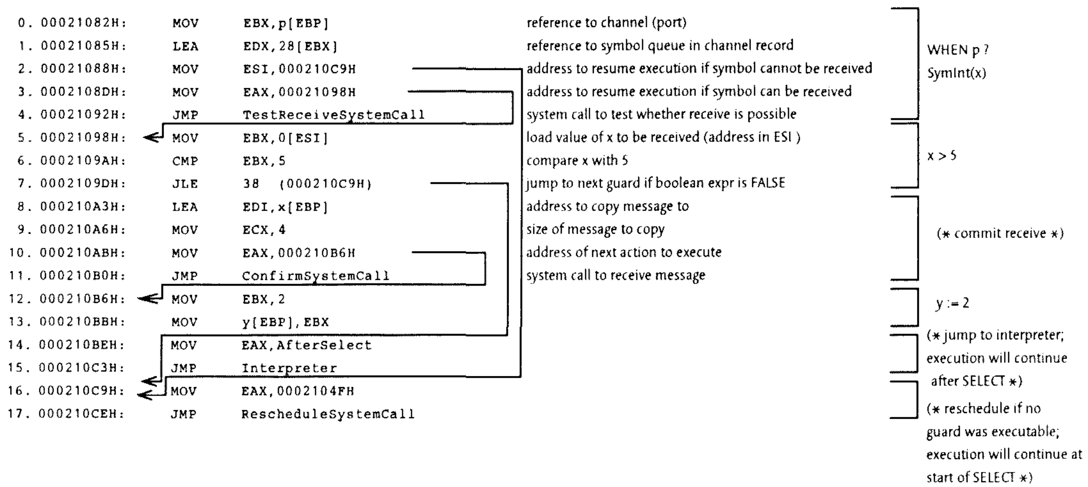

Figure 18: Intel 386 code generated for the second `WHEN` clause (containing a *Receive* guard) of the first `SELECT` example

- line 0: A reference to the channel over which the message will be sent is loaded into register `EBX`.
- line 1: A reference to the queue for this symbol (part of the data structure representing the channel) is loaded into `EDX`.
- line 2: The address of the action to execute if the signal cannot be sent (containing the `Reschedule` system call) is loaded into `ESI`.
- line 3: The address of the action to execute if the signal can be sent (the boolean expression `x > 5` in the guard) is loaded into `EAX`.
- line 4: The jump to the system call which examines whether communication with this guard is possible is executed.

If communication is possible, the boolean expression in the guard will be executed to determine whether communication should be done or not (lines 5–7). The boolean expression references a value to be received in the variable `x` (line 5). The address of this value is provided by the runtime system in register `ESI`, so this is why the value at `0[ESI]` is loaded into a register and compared to `5` (also refer to discussion in Section 4.3.1). If the boolean expression is `FALSE`, communication will not be done, and a jump is executed to the `Reschedule` system call (lines 16–17). If the boolean expression is `TRUE`, the second system call will be executed. This entails:

Stellenbosch University http://scholar.sun.ac.za

- line 8: The address where to copy the message to is loaded in register EDI.
- line 9: The size of the message to copy is loaded into ECX. The size is supplied by the sender and the receiver for more efficient implementation of the system call.
- line 10: The address of the next action to execute (the command y := 2) is loaded into EAX.
- line 11: The *Confirm Receive* call is executed. This will copy the message to the receiver.

If the message was received and the command y := 2 has been executed, a jump to the interpreter (lines 14–15) is executed. The next action will be the first action after the SELECT. If the message was not received, the Reschedule system call (lines 16–17) is executed; the next action will be the first guard of the SELECT.

Note that a *Send* guard will never need to address values to be received — there *are* no values to be received. Therefore a *Send* guard always contains only one system call, even if it contains a boolean expression. If a boolean expression is present, code for it is generated first, and if it is FALSE, execution will jump over the *Send* system call to the next guard.

A more intricate SELECT example

The guards in the SELECT discussed in Section 4.3.2 communicated only one value. However, LF allows transmission of more than one value per message. To illustrate this, we will discuss the first guard of the SELECT example given below:

```
SELECT
TYPE IntricateChanType = [SymDouble(INT32, CHAR), SymInt(INT32)];

PROCESS Server;
VAR x, y : INT32;
PORT p : IntricateChanType;
BEGIN
    (* ... *)
    SELECT
        WHEN p ! SymDouble(x + 3, ch) THEN y := 1
        WHEN p ? SymInt(x) & x > 5 THEN y := 2
    END
END Server;
```

In the first guard, two values are sent via port p. The first is an integer expression, and the second a character. Therefore data will be copied from two locations to the receiver. Two approaches are possible to send such multiple values. First, the process can copy all data to a central temporary record and generate a system call to transmit this one record. Likewise,

Stellenbosch University http://scholar.sun.ac.za

the receiver can accept the data into a similar record and copy all data from there to the necessary locations. The system call would be more efficient since a contiguous piece of data is copied; yet the complete operation would involve double copying.

Another approach would be to generate a table of addresses and sizes of all values to copy. We decided to use this alternative. For efficiency, the table can be generated by the compiler and placed inside the executable machine code. However, not all addresses of data can be resolved at compile time — examples of such data are expressions (calculated in registers by the LF compiler) and array elements indexed by variables. In our example, the first value to be sent with the `SymDouble` symbol is the expression `x + 3`. This value is saved to a temporary location in the activation record. The offset of this temporary location from the activation record base is known at compile time, and can be written to the table of addresses. The generated machine code for the example is shown in Figure 19.

The code in Figure 19 does the following:

- line 0 – line 3: The expression to be sent (`x + 3`) is calculated, and saved to the activation record.
- line 4 – line 9: The system call to attempt sending of the message is done. The following data is loaded into registers for the call:
  - line 4: The port over which the message will be sent is loaded into register `ECX`.
  - line 5: The base of the table of addresses and sizes is loaded into register `EBX`.
  - line 6: The reference to the symbol queue inside the channel record is loaded into `ESI`.
  - line 7: The address of the action to execute if this message cannot be sent is loaded into `EDX`.
  - line 8: The address of the action to execute if the message *can* be sent is loaded into `EAX`.

  After this, the call on line 9 is executed.
- line 10 – line 12: The table of sizes and addresses of data to copy is generated here inside the executable image. The table contains a 4-byte entry for the size followed by a 4-byte entry for the offset of the data relative to the process activation record base. In this example, the table specifies copying of a 4-byte value at offset 38H, and a 1-byte value at offset 30H in the activation record. The table is ended by a 4-byte value `0FFFF FFFFH`.

Stellenbosch University http://scholar.sun.ac.za

- line 13 – line 14: The assignment `y := 1` is done.
- line 15 – line 16: A jump to the interpreter is executed; the next action of this process will be the first action after the `SELECT`.

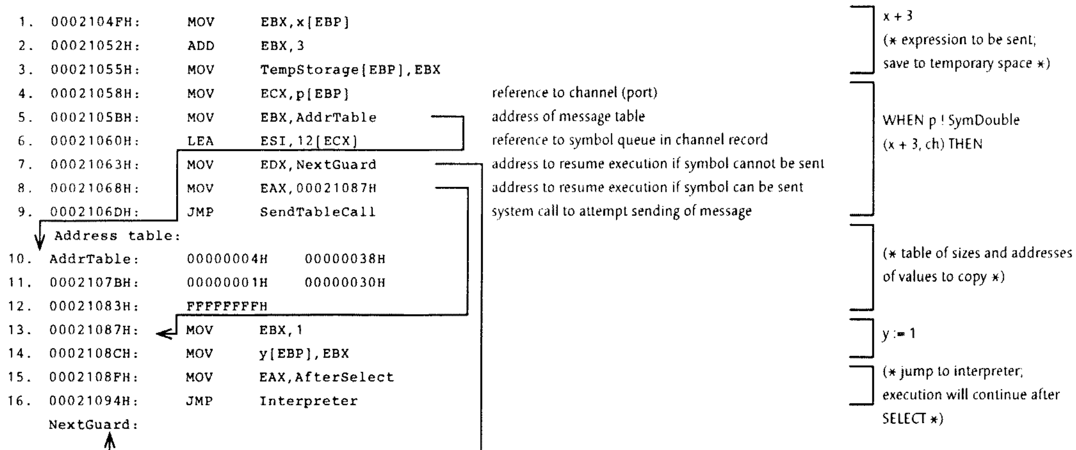

Figure 19: Intel 386 code generated for the first `WHEN` clause (containing a *Send* guard) of the second `SELECT` example

4.3.3 Process activation record layout

The process activation record (AR) contains the parameters and local variables of the process. In addition, the run-time system maintains information about the process inside the AR, such as the status of the process (for example executing, waiting for a receive, waiting for a send). If a register already in use is needed again (for multiplication, for example), it is saved to memory inside the activation record, so an area is also reserved to save every register once. A space is also reserved for all temporary variables needed, such as discussed in Section 4.3.2.

It is interesting to note that the code generated for an LF process does not maintain a dynamic link to the process which instantiated it. However, the run-time system maintains such links. This is so because:

- The run-time system is involved in instantiating and terminating processes, since processes are instantiated and terminated by executing system calls.
- When a process is terminated, execution does not continue in the instantiator of the terminated process as happens with procedures; in fact, each process has its own thread

Stellenbosch University http://scholar.sun.ac.za

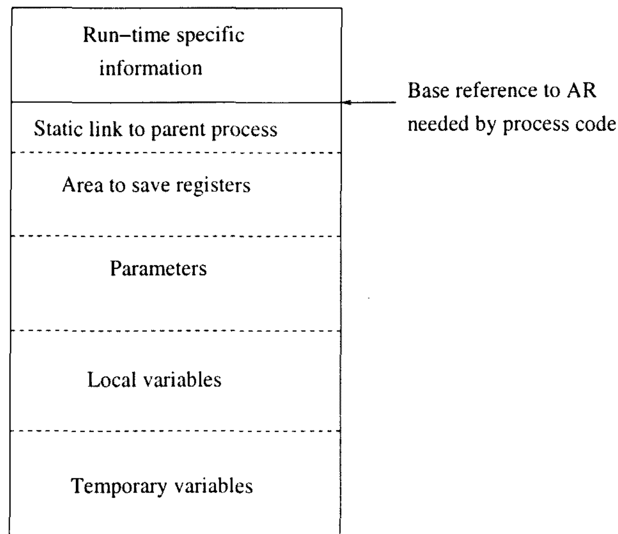

Figure 20: Organisation of memory inside a process activation record

of execution. However, a process can only terminate once all the processes instantiated by it has terminated, as discussed in Section 3.7.3. Therefore the run-time system needs to know which process has instantiated each terminated process.

- When a *called* process terminates, the caller of this process needs to be unblocked, as discussed in Section 3.6.1. The run-time system needs a link to the caller process to unblock it.

However, nested processes which are *called* are able to access the variables of processes (or modules) in which they are nested. In a static environment such as occam the compiler can resolve addresses of such global variables at compile time; however, in a dynamic environment such as LF, a static link to the process in which the *called* process is nested needs to be maintained. Processes reference global data in the following ways:

1. **A *called* process is visible globally in a module and it references variables global to that module.**

   Scoping rules in LF forbid a process referencing global variables from being exported. Therefore such a process can only be *called* from within the module where it is defined. The static link of such a process will refer to the module which *called* the process, so the compiler generates code to let the static link point to the AR of the caller.

2. **A process *calls* another process and the two processes are defined in the same scope.**

Stellenbosch University http://scholar.sun.ac.za

Therefore the two processes are defined within the same process or module, and the compiler can copy the static link from the caller to the callee.

3. **A process *calls* a process nested within itself.**

   Therefore the static link of the *called* process will refer to the caller; the compiler can let the static link refer to the AR of the caller, as in point 1.

4. **Ports defined globally to a module** may be exported; however, space for global ports are allocated in the image generated by the compiler, as described in Section 4.3.5. Therefore the compiler can resolve addresses of such ports at compile time; no static link is needed to resolve such references.

4.3.4 Process activation

The way that memory is allocated for processes in LF affects the efficiency of process instantiation. LF processes can run concurrently and are not guaranteed, like procedures, to terminate in any predictable order. Therefore the stack organisation of memory used in systems supporting procedures is not suited for a process-based language such as LF.

The allocation of memory for LF processes is based on a technique described in [9], and described as the *Quick Fit* allocation scheme. Memory blocks of fixed sizes are allocated to processes from a central memory pool, and when processes terminate, the memory is reclaimed into the central memory pool for re-use by other processes.

The Quick Fit scheme influences process instantiation because parameters have to be copied to the AR of a new process before the new process starts executing. In a stack-based architecture, a stack frame for a new procedure is allocated above or below the stack frame of the current procedure. However, in an LF system there is no way to predict at compile time where the new activation record will be located. The run-time system allocates memory, so a system call is needed to do this. However, once the memory has been allocated, arguments have to be copied to the new activation record. This can be done in several ways:

1. The compiler can copy all parameters to a buffer in its own activation record, and let the run-time system copy the parameters from there to the new activation record. Therefore, the data will be copied twice.

2. The compiler can compile a table of addresses and sizes of all the data to be copied, and the run-time system can copy all data to the new activation record. This technique is used to copy multiple values for interprocess communication, as described in

Stellenbosch University http://scholar.sun.ac.za

Section 4.3.2. It is considered efficient, because the table is compiled at compile time and forms part of the executable code.

3. The compiler can generate a system call to allocate memory for an AR, and the run-time system can supply a reference to the newly allocated memory in a register. The compiler can copy arguments to a location relative to this base reference, and activate the new process with an additional system call. Therefore two system calls are needed for this method. Also, no scheduling may take place between the two system calls, or the reference to the newly allocated memory may be overwritten.

   Because system calls were deemed reasonably efficient in the LF system, this last method was used to instantiate processes. Future experiments might find method 2 more efficient.

If the arguments can be transferred in the few remaining registers available on the 80386, a single system call is used to allocate memory, copy parameters and activate the process. The system calls are similar to those illustrated in Figure 15, 18, 17 and 19, and are therefore not shown here.

4.3.5 Modules and separate compilation

LF is a modular language, and is defined to support separate compilation of modules, and linking by a utility separate from the compiler. However, this version of the LF compiler performs linking of modules as well. When a module name is encountered in the `IMPORT` list, the compiler attempts to open a `.LF` source code file with the same name and compile it. Only if this is successful, will the parsing of the client module proceed. Because of this manner of operation, the module name and source file name of a module need to match.

Startup code allocates an activation record for each module and activates it as a process. However, as mentioned in Section 4.3.5, the module activation records stay allocated after the code of those modules have completed.

To perform linking, the compiler needs to be aware of the location of exported objects at compile time. In LF static information such as constants, types and process definitions can be exported, as well as ports. Constants, types and process definitions do not require run-time modifiable memory for storage. However ports (containing run-time memory references to channel activation records) do. The location of the activation record, where memory for variables and ports is normally reserved, will only be known at run-time. Therefore memory

Stellenbosch University http://scholar.sun.ac.za

for ports is not allocated as part of the activation record. Rather, space is left for ports in the compiled image, just after the machine code for that module.

For our experimental purposes, the linking compiler was deemed an adequate tool. Extensions and improvements can be made in future if necessary. In this way, the additional effort of defining an object file format, modifying the compiler to create these object files, and implementing a separate linker, has been avoided in this experiment.

```
TestModules.Dec   Close Hide Copy Search Rep RepAll Store        Style Tools
LF v2.0 -- Binary file decode

Jump to init code :
00021000H:E9 4A 02 00 00                   JMP     586  (0002124FH)

Module header for : Imported3
Code size : 00000006
Constant array size : 00000000
Global ports size : 00000000

00021031H:FF 25 B0 0F 00 00                JMP     [4016]  ; RTSC: ModuleToSleepCall


Module header for : Imported2
Code size : 0000003F
Constant array size : 00000008
Global ports size : 00000004

00021063H:BB A2 10 02 00                   MOV     EBX,000210A2H
00021068H:B9 01 00 00 00                   MOV     ECX,00000001H
0002106DH:BA 01 00 00 00                   MOV     EDX,00000001H
00021072H:BF 05 00 00 00                   MOV     EDI,00000005H
00021077H:B8 82 10 02 00                   MOV     EAX,00021082H
0002107CH:FF 25 6C 0F 00 00                JMP     [3948]  ; RTSC: WriteString
00021082H:BB 02 00 00 00                   MOV     EBX,00000002H
00021087H:B9 05 00 00 00                   MOV     ECX,00000005H
0002108CH:BA AA 10 02 00                   MOV     EDX,000210AAH
00021091H:B8 9C 10 02 00                   MOV     EAX,0002109CH
00021096H:FF 25 AC 0F 00 00                JMP     [4012]  ; RTSC: NewChannelCall
0002109CH:FF 25 B0 0F 00 00                JMP     [4016]  ; RTSC: ModuleToSleepCall
     0000:  54 79 70 65 3A 00 00 00                        Type:...
     0000:  00 00 00 00                                    ....


Module header for : Imported
Code size : 0000007D
Constant array size : 00000000
Global ports size : 00000000

000210DAH:BB 12 00 00 00                   MOV     EBX,00000012H
```

Figure 21: Part of a binary image output by the LF compiler, decoded.

Each imported module is fully compiled before the compiler proceeds with parsing of the importing module. This is to avoid having to maintain a potentially massive tree structure of several modules in memory.

An Oberon utility, Decoder.Mod, was modified to decode and disassemble the binary images generated by the compiler. Part of the output of the modified decoder, LFDecoder.Mod is shown in Figure 21. As can be seen, the first instruction is a jump to initialisation code. This initialisation code starts all modules as processes. It is appended at the end of the image when all modules have been compiled, and is shown in Figure 22.

The reader might notice that the system calls in Figure 21 and Figure 22 (for example at address `21263H` in Figure 22) are indirect jumps. In the illustrations earlier in the chapter

Stellenbosch University http://scholar.sun.ac.za


```
TestModules.Dec   Close  Hide  Copy  Search  Rep  RepAll  Store   Style  Tools
00021200H:8D 7D 28                   LEA    EDI,40[EBP]
00021203H:89 7B 30                   MOV    48[EBX],EDI
00021206H:8D 7D 2C                   LEA    EDI,44[EBP]
00021209H:89 7B 34                   MOV    52[EBX],EDI
0002120CH:B8 17 12 02 00             MOV    EAX,00021217H
00021211H:FF 25 C4 0F 00 00          JMP    [4036]  ; RTSC: CallProcessCall
00021217H:8B 5D 2C                   MOV    EBX,44[EBP]
0002121AH:89 5D 24                   MOV    36[EBP],EBX
0002121DH:FF 25 B0 0F 00 00          JMP    [4016]  ; RTSC: ModuleToSleepCall


Module header for: $MODULESTART
Code size : 0000006E
Constant array size : 00000000
Global ports size : 00000000

0002124FH:BB 31 10 02 00             MOV    EBX,00021031H
00021254H:BA 04 00 00 00             MOV    EDX,00000004H
00021259H:B9 24 00 00 00             MOV    ECX,00000024H
0002125EH:B8 69 12 02 00             MOV    EAX,00021269H
00021263H:FF 25 C0 0F 00 00          JMP    [4032]  ; RTSC: CreateProcessFewCal
00021269H:BB 63 10 02 00             MOV    EBX,00021063H
0002126EH:BA 03 00 00 00             MOV    EDX,00000003H
00021273H:B9 24 00 00 00             MOV    ECX,00000024H
00021278H:B8 83 12 02 00             MOV    EAX,00021283H
0002127DH:FF 25 C0 0F 00 00          JMP    [4032]  ; RTSC: CreateProcessFewCal
00021283H:BB DA 10 02 00             MOV    EBX,000210DAH
00021288H:BA 02 00 00 00             MOV    EDX,00000002H
0002128DH:B9 2C 00 00 00             MOV    ECX,0000002CH
00021292H:B8 9D 12 02 00             MOV    EAX,0002129DH
00021297H:FF 25 C0 0F 00 00          JMP    [4032]  ; RTSC: CreateProcessFewCal
0002129DH:BB 83 11 02 00             MOV    EBX,00021183H
000212A2H:BA 01 00 00 00             MOV    EDX,00000001H
000212A7H:B9 2C 00 00 00             MOV    ECX,0000002CH
000212ACH:B8 B7 12 02 00             MOV    EAX,000212B7H
000212B1H:FF 25 C0 0F 00 00          JMP    [4032]  ; RTSC: CreateProcessFewCal
000212B7H:FF 25 B4 0F 00 00          JMP    [4020]  ; RTSC: EndProcessCall
```

Figure 22: Part of a binary image output by the LF compiler, decoded.

(for example Figure 19) system calls and jumps to the interpreter are shown as direct jumps. While the LF system was in development, the memory organisation of the run-time system was changed frequently, so the addresses of all system calls were stored in a table. The processes jumped to the system calls indirectly via the addresses stored in this table. This mode of operation offers more flexibility, but is unnecessarily inefficient. The decoded images shown in Figure 21 and Figure 22 shows images generated by the compiler while in development mode. However, the assembler code illustrating the code generated for the various LF constructs, such as Figure 19, shows code generated for the final version of the system, where permanent memory locations for the interpreter and system calls have been determined.

4.4 Regression testing

During development of the compiler back-end, code generation routines were completed for one construct before beginning with the next. It was often necessary to modify and generalise existing code generation routines when writing new ones. Therefore, the risk was there that errors could be introduced into the existing routines.

Stellenbosch University http://scholar.sun.ac.za

To detect such errors, a utility was written to perform regression testing on the compiler. The utility operates as follows:

```
RegressReport.Text   Close Hide Copy Search Rep RepAll Store Style Tools
Failed:
Ex005.LF : failed, decode file pos 805
   Desktops.OpenDoc Ex005.LF~
   LF2Decoder.Decode Ex005~
   Desktops.OpenDoc Ex005.Right.Dec~

Ex005.LF : failed, tree report file pos 185
   Desktops.OpenDoc Ex005.Tr~
   Desktops.OpenDoc Ex005.Right.Tr~
Passed:
Ex005.LF : symbol table report correct.
LFTest0.LF : binary decode correct.
LFTest0.LF : tree report correct.
LFTest0.LF : symbol table report correct.
TableIPC.LF : binary decode correct.
TableIPC.LF : tree report correct.
TableIPC.LF : symbol table report correct.
TestRecord.LF : binary decode correct.
TestRecord.LF : tree report correct.
TestRecord.LF : symbol table report correct.
Test0.LF : binary decode correct.
Test0.LF : tree report correct.
Test0.LF : symbol table report correct.
Test1.LF : binary decode correct.
Test1.LF : tree report correct.
Test1.LF : symbol table report correct.
Test3.LF : binary decode correct.
Test3.LF : tree report correct.
Test3.LF : symbol table report correct.
Test4.LF : binary decode correct.
Test4.LF : tree report correct.
Test4.LF : symbol table report correct.
Test5.LF : binary decode correct.
Test5.LF : tree report correct.
Test5.LF : symbol table report correct.
TestParms.LF : binary decode correct.
TestParms.LF : tree report correct.
TestParms.LF : symbol table report correct.
```

Figure 23: Report generated by the regression test utility

- A series of example LF programs are written and compiled.
- The output from the decoder utility described in Section 4.3.5, LFDecoder.Mod is saved.
- Text-format reports about the symbol table and tree are also saved.
- The names of all the LF test modules are saved in an input file which the regression test utility uses as input.
- When the test utility is executed, the compiled image of every LF file in the input file is deleted, if present. The file is compiled and the output decoded, and then compared to the saved decoded file. The symbol table and tree reports are also compared.
- Any textual difference is reported at the end of the regression test, as is shown in Figure 23.

The advantage of a regression test system as described above is that it is automated. After completing extensions or modifications to the compiler, one can ensure that no bugs were

Stellenbosch University http://scholar.sun.ac.za

inadvertedly introduced into the compiler simply by running the test suite. The disadvantage is that much time is needed to maintain and extend this test suite as development progresses. However, time spent on such an activity will lessen the time spent tracing and removing errors after development. Several errors which were introduced when extending compiler functionality were found with this system.

4.5 Overhead of the interpreter

An important question to be answered is how much overhead will be added to a system if it is interpreted as discussed in this chapter. Each time that the interpreter is executed, at least the following instructions will be executed (the pseudocode for the interpreter is shown in Figure 3):

```
  MOV   ESI, adrInterrupts
  CMP   DWORD [ESI], 0      ; IF Interrupts # {} THEN
  JZ    CONTINUE
CONTINUE:                   ; END;
  JMP   EAX                 ; Continue(); (* execute next action *)
```

This piece of code:

- examines if interrupts have occurred,
- if no interrupts are marked as having occurred, control is transferred to the end of the interpreter
- control is transferred to the next action of the current process

If interrupts *have* occurred, the last `JMP` in the interpreter will not be executed, since the scheduler does not return to the interpreter. However, the scheduler contains a similar jump which also transfers control to the next action of the current process, as discussed in Section 4.2.1. The interpreter will also be executed when a system call is done; this is done to service interrupts and select the next action to execute.

The version of the LF compiler documented here generates code for the Intel 80386 architecture which is commonly used for embedded applications. In contrast with later Intel architectures, Intel still published the number of clock cycles that 80386 instructions take to

Stellenbosch University http://scholar.sun.ac.za

execute. However, these figures depend on a few assumptions. For example, the instruction must already have been prefetched and decoded, and must be ready for execution.

This assumption is invalidated by a piece of code executing a jump instruction because the 80386 does not implement branch prediction algorithms. Therefore the first instruction to be executed after a jump has not been prefetched. Since control is transferred to the LF interpreter via a jump instruction, and the interpreter does contain other jumps as well, it is difficult to calculate how many clock cycles will be needed to execute the interpreter.

Direct measurement of the overhead can also be misleading, since the amount of overhead depends on the length of the actions. Several consecutive assignments are composed into a single action by the compiler. Code for an `IF` construct executes one jump to the interpreter, as long as the body of the `IF`, `ELSIF` or `ELSE` clause which is executed does not contain other jumps to the interpreter (see Section 4.3.2). Code for a `WHILE` construct (also discussed in Section 4.3.2) contains one jump to the interpreter per iteration.

Rough measurements of the overhead were taken for five pieces of LF code. This was done by measuring the time taken to execute the same piece of code with and without jumps to the interpreter, and comparing the two different times. These measurements show how the overhead varies depending on the control flow structure of the code. The five pieces of code are shown below, and the overhead for each of the pieces of code is shown in Table 4.5.

```
(* code for test 1 *)
WHILE x < 20000000 DO x := x + 1 END ;

(* code for test 2 *)
WHILE x < 20000000 DO
    x := x + 1; x := x; x := x; x := x; x := x;
    x := x; x := x; x := x; x := x; x := x; x := x
END ;

(* code for test 3 *)
WHILE x < 20000000 DO x := x + 1;
    IF x > 10000000 THEN x := x END
END ;

(* code for test 4 *)
WHILE x < 20000000 DO x := x + 1;
    IF x > 10000000 THEN
        x := x;
        IF x = 5 THEN
            x := x - x + x;
            IF x = 10 THEN x := x END
        END
    END
END ;

(* code for test 5 *)
WHILE x < 5000000 DO
    buf[x MOD BufferSize].name := "John Smith";
    buf[x MOD BufferSize].payload := "Empty";
```

Stellenbosch University http://scholar.sun.ac.za

Measurement of overhead for LF code samples			
Code piece	*Execution time with interpreting*	*Execution time without interpreting*	*Overhead*
Test 1	1min 37.5s	45.5s	114%
Test 2	3min 59.0s	3min 13.0s	23%
Test 3	3min 59.5s	1min 22.9s	169%
Test 4	5min 08.5s	1min 38.3s	213%
Test 5	3min 00.6s	2min 25.5s	24%

Table 1: Overhead of the interpreter for five test cases

```
    IF x < 1000 THEN
        buf[x MOD BufferSize].num := x*5 DIV 100 + 15000*x
    ELSE buf[x MOD BufferSize].num := x*5 DIV 100
    END ;
    x := x + 1
END ;
```

In the first test case, a single assignment is executed for every iteration. This assignment is similar to the assignments shown in Section 4.3.2 which consists of `MOV` instructions to load the values and save the expression to the variable. In this assignment, an `ADD` instruction will also be used to calculate the expression `x + 1`. The interpreter is also executed in every iteration. Overhead of interpreting consists of a `JMP` to and from the interpreter, as well as three instructions if no interrupts have occurred, and more if interrupts have occurred. The number of instructions executed for interpretation is therefore at least roughly as many, and probably more, than the number of instructions executed for the assignment, so the overhead of 114% is to be expected for the first test case.

In the second test case, the number of data manipulations (assignments) are many compared to the intricacy of the control flow (a single `WHILE` loop). This is why the overhead is drastically lower here (23%) than in the first example. In the third and fourth example, the control flow is intricate compared to the single assignments, with very high overhead as result.

The fifth test case shows a piece of code with control flow structure which is typically found in implementations. However, the overhead of 24% cannot be assumed to be typical, as the previous examples illustrate by how much the overhead can vary.

The tests were conducted on an 80386SX running at 25MHz, and the measurements were taken with a stopwatch. The measurements were repeated 5 times for each test and the time noted in the table is the average of those times. The times measured was roughly the same for every test — the variance which was observed was attributable to the imprecise method

Stellenbosch University http://scholar.sun.ac.za

of measuring time.

The rough measurements serve to illustrate that the Oberon-type control flow which is present in LF affects efficiency of transition systems negatively. Two approaches can be taken to compose several high-level commands into one action. Because such work falls outside the scope of this thesis, neither approach was investigated; however, both are discussed in Section 5.3.1.

4.6 Summary

Details about the implementation of the LF compiler and the transition system generated by it were presented in this chapter. It entailed the following:

- The design goals of the compiler and its output is outlined in Section 4.1.
- In Section 4.2 is explained what is understood under the term *transition system*. A short introduction to assembly language for the Intel 80386 processor is given. A version of the 80386 called the 80386EX is often used for embedded applications, and is the target processor for the compiler back-end. An overview is given of the interpreter which executes the transition system.
- The compiler is discussed in Section 4.3. Aspects examined are the symbol table, syntax tree structure, code generation, and modules and linking.
- When code is executed as a transition system by an interpreter, some overhead is incurred. In Section 4.5 we showed how performance was impaired by the transition system approach.

Stellenbosch University http://scholar.sun.ac.za

Chapter 5

Conclusion

In Chapter 1 it was stated that reliability was an important consideration for embedded systems. An environment which could make it easier to develop reliable embedded systems was envisioned, and it was stated that the experimental language and compiler discussed in this thesis is intended to form part of such a system. The language and compiler were to assist in detecting programming errors at compile time. The language and form of the executable code were also to be adapted to make model checking of implementations feasible.

Below, LF is reviewed in the light of the design goals outlined for the language, and compared to other languages similar in design. The influence of executable code generated as transition systems is also discussed.

5.1 Revision of design goals for the language

In section 3.2 several design goals were set out for the LF language. Ways were described in which the LF language could promote reliability — an important aim was to generate implementations for which model checking would be practical. The language is also intended to support the implementation of embedded systems, so aspects which would assist embedded development were highlighted. The design goals were discussed in more detail under the following headings:

- **CSP as design framework** CSP does not allow concurrently executing processes to share data, and the only way in which processes can communicate is by synchronous message passing. The concurrency model of LF is based on that of the CSP-based

Stellenbosch University http://scholar.sun.ac.za

language Joyce, which also supports both private data for processes and synchronous message passing. However, the restriction on sharing of data in Joyce has been relaxed so that a process which is *called* (the instantiator stays blocked while the *called* process completes) can access the data of its instantiator. This measure is inspired by the rules for variable access in another CSP-based language, occam.

- **Eliminate language features which complicate model checking** Pointer support is an example of a language feature normally viewed as important for implementation of efficient embedded systems — pointers can be used to optimise memory usage when many variables of different types are needed. However, in many cases dynamic data structures can be implemented as arrays — this is frequently done for embedded systems, and is preferable for real-time systems. Also, occam illustrated that non-trivial systems can be implemented without pointers [13].

  Therefore pointers were not supported in this experiment. As a result, memory usage will sometimes be less efficient. However, hardware for embedded systems, including memory, is becoming more powerful and less expensive, while the cost of developing reliable software stays relatively high. A tool which could reduce cost of development and testing at the price of higher memory requirements is therefore becoming relevant.

- **Safe programming practices** It was stated that by encouraging safe programming practices, reliability of software can be enhanced. In section 3.2 strong typing was highlighted as a feature which could encourage safe programming practices. Strong type checking was implemented for parameter passing, assignments and interprocess communication. Other language and compiler features which prevent or detect programming errors are disjoint data areas for concurrently running processes (preventing processes from corrupting data of other processes) and run-time checks to detect out-of-bounds array indexing and overflow on arithmetic operations.

- **Intuitive and easy to understand language** Some guidelines to make the language intuitive and clear were outlined. For example, notations and constructs in LF which resemble those encountered in conventional languages should be implemented as programmers know them. Examples of such features in LF are control flow constructs, parameter passing and scoping, which are implemented as in Algol-like programming languages. The language is small compared to a language such as C++ or even occam 3, also making it easy to understand.

- **Small runtime system** The runtime system needed is small and simple, but some complications are deemed justified.

Stellenbosch University http://scholar.sun.ac.za

An efficient scheduler and efficient message passing support are needed, as stated in section 3.2. However, the LF concurrency model is based on Joyce, which requires more run-time support than, for example, occam. The sharing of channels requires processes waiting to communicate to be queued by channels. Support for dynamic allocation of process memory and channels had to be implemented. A feature such as the selective receive implemented for `SELECT` guards complicates the compiler and the run-time system.

However, all of these features offer advantages. Sharing of channels simplifies interfaces between processes. Dynamic process creation and channel allocation provide more flexibility in LF compared to the static nature of occam. For example, creating a client-server architecture is much more convenient in LF than in occam, where a server needs a channel for every client The selective receive in the `SELECT` provides a facility that the user would have had to implement otherwise.

- **Low-level operations** Examples of LF features which make development of embedded implementations more convenient are:
  - The `AT` construct to place variables at specific memory addresses.
  - The `PORTIN` and `PORTOUT` intrinsic processes to write to hardware ports.
  - Operations on sets for bit manipulation.
  - Interrupt channels to allow device drivers to be implemented as LF processes outside the runtime system.

  Because data can be shared between a *called* process and its instantiator, less copying is necessary in an LF implementation than in Joyce, making the system more efficient.

- **Context switching and interprocess communication in software** The language constructs and concurrency model used in LF is less fine-grained than that of occam. Since processes normally contain more commands in LF, less context switching and interprocess communication is needed than in occam, which makes LF more suitable for conventional embedded architectures. An added advantage of LF is that, because the language implements disjoint data areas for concurrent processes, no runtime memory protection is needed. Data transfer between processes can therefore be implemented more efficiently.

Stellenbosch University http://scholar.sun.ac.za

5.2 The influence of the transition system approach

LF executable code are generated as a transition system, to assist in model checking efforts. The transition system has the following influences:

- Points where scheduling and servicing of interrupts can take place in a process are limited. Therefore, the ways in which process code can be interleaved during execution are limited; the system will become less complex and the state space of the system for model checking will be reduced.
- The compiler must generate code into actions, and must also limit the duration of such actions. This complicates code generation.
- No registers can be in use between actions, so future optimisation efforts will have limited opportunities to optimise register usage.
- The current compiler generates transition systems with too short actions, causing considerable overhead. To reduce this overhead, either the language or the compiler will have to be adapted to make actions longer.

5.3 Future work

Many different areas for future work still exist in LF — the most significant of these areas are the optimisation of the generated transition systems.

5.3.1 To improve the efficiency of the transition system

As stated in section 4.1, interrupts are only serviced between actions; therefore the action should be made short enough so interrupts are not missed. However, in section 4.5 it was observed that the control flow constructs in LF usually limit actions to much shorter lengths than needed, increasing the overhead unnecessarily. Therefore a transition system could be far more efficient than described in section 4.5. Two ways to lengthen the actions in an LF transition system are described below. For both ways, the user or system designer should specify the time limit on actions. This will probably be slightly shorter than the minimum possible time between the occurrence of the same interrupt. After comparing these two methods, some other methods of improving efficiency of the LF system are reviewed.

Stellenbosch University http://scholar.sun.ac.za

Adapt the language

The language can be adapted so the compiler can compose more constructs into actions. For example, consider the short-circuit evaluation of boolean expressions, which is already implemented in LF. Because of this feature, fewer nested `IF` constructs are needed in an LF program. For example, if a programmer needs to ensure that a variable a is non-zero before calculating b div a, he or she can use the composite expression `(a # 0) & (b DIV a)` instead of two nested `IF`s.

A second example is a `WHILE` construct with several `IF`s nested within. This breaks execution into more actions than needed. Instead, the language could include a loop containing guarded commands. The compiler can then determine the guarded command which would take the longest to execute, and ensure that *that* execution path does not lead to a too long action. In this way, one iteration of the loop can be a single action with several possible execution paths.

A loop with guarded options might have the same semantics as the repetitive construct described in [10]. Such a loop might look as follows in LF:

```
DO
WHEN ch1 ? sym1(x) THEN y := 1
WHEN ch2 ? sym2(x) THEN y := 2
WHEN ch3 ? sym3(x) THEN y := 3
END
```

However, note the discussion in [10, Paragraph 7.9] about the termination of such a loop; if the guards of the loop contain input commands, the loop may only terminate once all possible sources of such input commands have terminated. If such a command were implemented, the run-time system would be severely complicated.

A possible way to avoid such complications would be to disallow communication in the guards of such a construct. This is reasonable, since the `SELECT` already implements guarded commands *with* guards.

Improve the compiler

Programs written in a language adapted as described above, will contain much less nested control flow structures. However, such nested structures will never be absent in a program. To compile the most efficient transition systems, compiler optimisations should be implemented.

Stellenbosch University http://scholar.sun.ac.za

The compiler will need to group several commands into one action. This can be done in the following way:

- The compiler must analyse the flow of execution and find the execution path which would take the longest. To do this, the worst-case execution times of constructs need to be calculated by the compiler.
- A control flow graph of the LF program must then be built, and annotated with the worst-case execution times.
- The control flow graph can be used to find the execution paths which would take the longest. If such paths exceed the maximum time allowed for an action they can be divided into as many actions as needed. If not, the longest path can be compiled into a single action together with all shorter paths possible.

This would complicate the implementation of the compiler. However, actions would be longer, while all interrupts would still be guaranteed to be serviced.

Other ways to improve efficiency

All actions in the transition system currently jump to a single routine which examines whether or not interrupts have occurred. However, long periods of time might pass where no interrupts have occurred. By including this examination in actions, and only jumping to the code servicing interrupts when there *have been* interrupts, many unnecessary jumps can be avoided. This inlining of code will cause an increase in code size; however, the increase will become less severe as the length of actions are increased. However, the implications of such a modification for model checking have to be considered.

Processes are blocked when attempting to send or receive a message over a channel and no communication partner is available on that channel. However, a process executing a `SELECT` currently polls all channels over which messages could be communicated. In large systems, the overhead involved in this polling will be significant. If the runtime system could block processes in a `SELECT` until a suitable communication partner is found, this overhead would be eliminated.

Stellenbosch University http://scholar.sun.ac.za

5.3.2 Other improvements

Avoid processes 'interfering' in the variables of each other (creating data race conditions) in a less restrictive way (see section 4.3.1). For example, consider a nested process that accesses a variable declared in the parent. Currently, this nested process can only be *called*, not *created*. To improve, allow the parent to *create* the nested process, but then disallow the parent from ever accessing the shared variable again.

5.4 Final thoughts

Much work needs to be done before the LF system can be used to develop more correct embedded software. Only when the language, compiler and run-time system have been used to implement real embedded systems, can be determined how suitable the tools are for the intended purposes. A model checker to verify the executable code still needs to be implemented, and other tools such as a remote debugger can further assist development of more correct embedded software.

However, the language and compiler are important tools to assist such an endeavour. Many features in LF have been inherited from other languages, where these features have been proven successful in assisting development of reliable software. No single solution to the problem of error-prone software exists, so the best way to provide reliability is to use a variety of tools and techniques together.

Stellenbosch University http://scholar.sun.ac.za

Appendix A

EBNF of LF

```
Module = "MODULE" Id ";" Import Declarations Body "."
Body = "BEGIN" [CommandList] "END" Id
Import = "IMPORT" Id {"," Id} ";"

DECLARATIONS
Declarations = { "CONST" Id ["*"] "=" Expr ";" {Id "=" Expr ";"} |
  "TYPE" Id ["*"] "=" Type ";" {Id "=" Type ";"} |
  "VAR" VariableDef ";" {VariableDef ";" ["AT" Number]} |
  "PORT" VariableDef ";" {VariableDef ";"} |
  Process }
Process = "PROCESS" Id ["*"] [ParmList] [":" TypeIdent] ";"
  Declarations Body ";"
ParmList = "(" ["VAR"] VariableDef { ";" ["VAR"] VariableDef } ")"
Type = "ARRAY" Expr "OF" Type | AlphabetDef |
  "RECORD" FieldList "END" | TypeIdent
FieldList = Id {"," Id} ":" TypeIdent {";" Id {"," Id} ":" TypeIdent}
AlphabetDef = "{" Symbol {"," Symbol} "}"
Symbol = Id ["(" TypeIdent {"," TypeIdent}")"]
VariableDef = Id {"," Id} ":" TypeIdent
TypeIdent = QualId

STATEMENTS
CommandList = Command {";" Command}
```

Stellenbosch University http://scholar.sun.ac.za


```
Command = [ If | While | Repeat | Select | Create | Call | Access |
  Return ]
Access = QualId Selector AssignOrIO
AssignOrIO = Assign | Send | Receive
Assign = ":=" Expr
Send = "!" Id ["(" Expr {"," Expr} ")"]
Receive = "?" Id ["(" QualId Selector {"," QualId Selector}")"]
Repeat = "REPEAT" CommandList "UNTIL" Expr
While = "WHILE" Expr "DO" CommandList "END"
Create = "CREATE" QualId ["(" Expr {"," Expr} ")"]
Call = QualId ["(" Expr {"," Expr} ")"]
If = "IF" Expr "THEN" CommandList
  {"ELSIF" Expr "THEN" CommandList}
  ["ELSE" CommandList] END
Case = "CASE" Expr "OF" CaseClause {"|" CaseClause}
  ["ELSE" CommandList "END" ]
CaseClause = [CaseLabelList ":" CommandList]
CaseLabelList = CaseLabels {"," CaseLabels}
CaseLabels = Expr [".." Expr] (* constant expressions *)
Return = "RETURN" [Expr]
Select = "SELECT"
  "WHEN" SelectGuard "THEN" CommandList
  {"WHEN" SelectGuard "THEN" CommandList} "END"
SelectGuard = QualId Selector SelectIO ["&" Expr]
SelectIO = Send | Receive

EXPRESSIONS
Selector = {"[" Expr "]" | "." Id }
Expr = Primary {PrimaryOp Primary}
PrimaryOp = "&" | "OR"
Primary = Secondary {SecondaryOp Secondary}
SecondaryOp = "<" | "<=" | ">" | ">=" | "=" | "#"
Secondary = Term {AddingOp Term}
AddingOp = "+" | "-"
Term = Factor {MultiplyOp Factor}
MultiplyOp = "*" | "DIV" | "MOD"
```

Stellenbosch University http://scholar.sun.ac.za


```
Factor = Number | "TRUE" | "FALSE" | "~" Expr | "-" Expr |
  "(" Expr ")" | QualId Selector | CharConst

TOKENS
QualId = Id ["." Id]
Id = Letter {Letter | Digit}
Number = ["$"] Digit {Digit | HexDigit}
Letter = "a".."z" | "A".."Z"
HexDigit = "a".."f"
Digit = "0".."9"
CharConst = Letter
```

Stellenbosch University http://scholar.sun.ac.za

Appendix B

Intrinsic processes

NEW, PORTIN and PORTOUT were the first examples of processes defined intrinsically in the language; the full list of these processes, which can only be *called*, is given below.

Intrinsic function processes			
Name	*Argument Type*	*Result Type*	*Function*
ABS(v)	signed integer type	type of v	absolute value
ASH(x, n)	integer type	INT32	$x2^n$

Intrinsic proper processes		
Name	*Argument Type*	*Function*
INC(v)	integer type	v := v + 1
DEC(v)	integer type	v := v - 1
NEW(v)	channel type	allocate a new channel
PORTIN(adr, v)	integer types	read from memory-mapped IO port
PORTOUT(adr, v)	integer types	write to memory-mapped IO port

Stellenbosch University http://scholar.sun.ac.za

Bibliography

[1] André Arnold. *Finite Transition Systems.* Prentice-Hall, 1992.

[2] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[3] Geoff Barrett. *occam 3 reference manual.* Inmos Limited, 1992.

[4] Dirk Bull. A Comparison of Two Different Model Checking Techniques. Master's thesis, University of Stellenbosch, 2003.

[5] Intel Corporation. *Intel Architecture Developer's Manual.* Intel Corporation, 1997.

[6] Régis Crelier. OP2: A Portable Oberon Compiler. Technical Report 125, Eidgenössische Technische Hochschule Zürich, February 1990.

[7] D. Crookes and J. W. G. Elder. An Experiment in Language Design for Distributed Systems. *Software — Practice and Experience*, 14(4):957–971, April 1984.

[8] Per Brinch Hansen. Joyce — A Programming Language for Distributed Systems. *Software — Practice and Experience*, 17(1):29–50, January 1987.

[9] Per Brinch Hansen. Efficient Parallel Recursion. *ACM SIGPLAN Notices*, 30(12):9–16, December 1995.

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

[11] G. J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[12] G. J. Holzmann. *The Spin Model Checker — Primer and Reference Manual.* Addison-Wesley, 2004.

Stellenbosch University http://scholar.sun.ac.za

[13] Geraint Jones. *Programming in occam.* Prentice-Hall, 1987.

[14] Geraint Jones and Michael Goldsmith. *Programming in occam 2.* Prentice-Hall, 1988.

[15] Nancy Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[16] SGS-Thomson Microelectronics Limited. *occam 2.1 reference manual.* SGS-Thomson Microelectronics Limited, 1995.

[17] David May. OCCAM. *SIGPLAN Notices*, 18(4):69–79, April 1983.

[18] T. J. Roper and C. J. Barter. A Communicating Sequential Process Language and Implementation. *Software — Practice and Experience*, 11:1215–1234, 1981.

[19] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts.* Wiley, fifth edition, 1997.

[20] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[21] Frank van Riet. LF: A Language for Reliable Embedded Systems. Master's thesis, University of Stellenbosch, 2001.

[22] J. Welsh, W. J. Sneerigner, and C. A. R. Hoare. Ambiguities and Insecurities in Pascal. *Software — Practice and Experience*, 7:685–696, May 1977.

[23] N. Wirth and J. Gutknecht. The Oberon System. *Software — Practice and Experience*, 19(9):857–893, September 1989.

[24] Niklaus Wirth. *Compiler Construction.* Addison-Wesley, 1996.

[25] Niklaus Wirth and Martin Reiser. *Programming in Oberon.* Addison-Wesley, 1992.