John Billings
Queens' College
jnb26

Computer Science Tripos Part II Dissertation

A Bytecode Compiler for Acute

2005

John Billings, Queens' College

A Bytecode Compiler for Acute

Computer Science Tripos, Part II, 2005

Word Count	11,960
Project Originator	Dr Peter Sewell
Project Supervisor	Dr Peter Sewell

Project aims

The aim of this project was to increase the performance of the Acute programming language by compiling it into O'Caml bytecode. We were unsure whether many of the new facilities for distributed programming were amenable to an efficient, low-level implementation. This project has direct implications on the future use of the Acute features in production quality languages.

Work completed

A bytecode compiler was implemented for a fragment of the Acute language. The O'Caml virtual machine was then extended in order to synthesise the novel marshalling primitives. Subsequently, we implemented basic modules and rebinding for marshalled values. This permitted a non-trivial, distributed example to be explored.

Special difficulties

None

Declaration of Originality

I John Billings of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

1.1 Background

Distributed computation is a hard problem. It is remarkable that the Internet functions at all. Servers crash and network connections fail. And all too often the component systems interact in unforseen ways. Bug-fixes are commonplace, causing the concurrent deployment of multiple versions of software. Ad-hoc methods are used to transfer code and data. Mainstream programming languages, including ML, Java and C$^\sharp$, provide little help for the systems designer.

The Acute language, developed by researchers at Cambridge and INRIA Rocquencourt, addresses some of these problems. It extends the core O'Caml language to provide facilities such as type-safe marshalling, dynamic linking and rebinding. These features permit distributed infrastructures to be built as simple libraries. Furthermore, the language is backed by a clean semantic model.

1.2 Project aims

The Acute language currently runs under an interpreter. The compiler emits an intermediate language, which is essentially the abstract syntax of Acute. The runtime then performs reductions over this language form. Whilst this simple approach to program execution is useful for clarifying the operational semantics, the execution speed is typically several orders of magnitude slower than that of the O'Caml runtime.

The aim of this project was to increase the performance of a fragment of Acute by compiling the intermediate language into a more efficient, low-level format. We chose to target O'Caml bytecode due to the similarities between Acute and O'Caml. Additionally, the O'Caml virtual machine offers a high performance, cross-platform execution environment.

From the outset of this project it was clear that the proposed work would require extending the O'Caml runtime in order to synthesise the novel Acute primitives. However, it was not obvious whether many of these new facilities for distributed programming were amenable to an efficient, low-level implementation. This work therefore has direct implications on the future use of the Acute features in production quality languages.

1.3 The Acute language

We shall assume some familiarity with the ML language. The O'Caml syntax that Acute extends is similar to ML and hence should be easily understandable.

The main experimental part of this project concerned the integration of the Acute marshalling primitives into the O'Caml runtime. In outline, marshalling allows the interconversion between arbitrary values and byte-strings. This section contains a brief overview of marshalling as well as the interactions between modules and rebinding. Further information is contained within the Acute Technical Report [13].

1.3.1 Marshalling

Suppose that we have just entered the code for the successor function into the interactive Acute top-level interpreter[1]. We might then wish to send this function value to another runtime by using the `IO.send` function of language type `string -> unit`. We must therefore first convert our successor function into a string value by using the `marshal` keyword.

```
let succ n = n + 1 in
let s = marshal "MK" succ in
IO.send s
```

From within a separate Acute interpreter we invoke the `IO.receive` function to retrieve the string value. Using the `unmarshal` keyword, we 'reanimate' the original function value. Applying this function to the integer value `42` yields `43` – our successor function has been successfully copied to this second runtime.

```
let s = IO.receive () in
let succ = unmarshal s as (int -> int) in
succ 42
```

Temporarily ignoring the use of the `MK` value during marshaling, we note that the unmarshalling primitive is supplied with the expected value type of `int -> int`. This is necessary to permit type-checking. If the unmarshalled value is of a different type then the runtime raises an exception.

[1] For simplicity, we give the corresponding fragments of compilable Acute

1.3.2 Modules and rebinding

In the general case we might wish to marshal a function that references module values. Clearly we can expect some modules to be present on all systems – perhaps they are part of our software distribution, or they might even be packaged with the language runtime. Marshalling such modules would be wasteful of resources. Module `M1` below shall assume this ubiquitous role. However, other modules, such as `M2`, may have only been recently developed or updated. We cannot guarantee that these modules will be present at unmarshal time. Therefore, should such module definitions be referenced by a marshalled value, then they must also be marshalled.

In this second example, we see Acute's linear sequence of marks and modules. An Acute program simply consists of a sequence of zero or more marks and modules, followed by at least one process. The mark `MK` is used to partition the modules `M1` and `M2`. Although the marshalled function `f` references both `M1` and `M2` (via module fields `M1.x` and `M2.y` respectively), only `M2` occurs after the marshal mark `MK`. Therefore we only marshal module definition `M2` along with `f`.

```
module M1 : sig val x:int end = struct let x = 17 end
mark "MK"
module M2 : sig val y:int end = struct let y = 42 end
let f () = M1.x + M2.y in
let s = marshal "MK" f in
IO.send s
```

In our second runtime, module `M1` is indeed present. Upon unmarshalling `f` we simply *rebind* the `M1.x` field reference to this module definition. However, as anticipated, module `M2` is absent. The runtime adds the included definition to end of the mark/module list for the use of `f`. Applying `f` to the unitary value results in the expected integer value of `59`.

```
module M1 : sig val x:int end = struct let x = 17 end
let s = IO.receive () in
let f = unmarshal s as (unit -> int) in
f ()
```

This rebinding facility means that Acute cannot use standard *call-by-value* operational semantics. This is because we need to preserve module references in order to permit later rebinding. Therefore Acute uses a *redex-time* variant for module references, where the values are only instantiated when they occur in *redex-position.* Standard *call-by-value* operational semantics are maintained for local expression reduction. In the above example, both `M1.x` and `M2.y` are redex-protected because they occur within the body of the marshalled function.

The Acute language also allows the developer to specify minimum versions of modules to which an unmarshalled value may rebind. This facility becomes invaluable when propagating updates to a distributed system. Although we do not further explore this facility during the project, we do lay the groundwork for such functionality.

1.4 Related work

The O'Caml runtime does itself support a limited form of marshalling. However, there is no guarantee of type safety. Furthermore, no actual code transfer occurs; *bytecode is simply marshalled as a code pointer and therefore can only be read back in a process running exactly the same code.* This would have prevented us from transferring our successor function to the second runtime. The Acute language overcomes this severe limitation.

The JoCaml language [7] extends the O'Caml runtime, adding support for the distributed join-calculus programming model. The *location* and *channel* facilities operate at a considerably higher level of abstraction than the Acute primitives, although they may be programmed as an Acute library. Few of the implementation details are published.

The Java language [1] permits the type-safe marshalling of arbitrary values through the use of serialisation. However, these facilities are class-based rather than directly integrated into the language. Furthermore, in contrast to Acute, there is no language-level support for versioning (with the exception of class serial numbers) or for the controlled rebinding of object references. Whilst Acute permits the marshalling of threads, there is no such facility in Java. Combined with the lack of a clean semantic model for object marshalling, these shortcomings hinder the development of distributed systems under Java.

Chapter 2

Preparation

2.1 Overview

This project was broadly divided into three stages. Initially, the back-end of the Acute compiler was modified to output O'Caml bytecode object files. We then extended the O'Caml runtime to support the marshalling of integer and function values. Finally, as an ambitious extension, we enhanced the compiler and runtime to support modules and rebinding.

In this chapter we detail the requirements and development methods used throughout the project. We then explore the underlying principles used in the design of the bytecode compiler, before proceeding onto an introduction into the operation of the O'Caml virtual machine. This latter section becomes relevant when we consider the runtime extraction of bytecode during the marshalling of function values.

2.2 Requirements

A requirement is 'a condition or capability needed by a user to solve a problem or achieve an objective' [2]. The high-level requirements analysis for a compiler of a well-specified language is relatively straightforward. The compiler must accept a given grammar and output valid code for the specified target architecture. Based upon the acceptance criterion ventured in the project proposal and considerable additional research, the following set of requirements was formulated for the compiler and runtime extensions:

- The compiler must accept the shaded terms from the source grammar specified in Figure 2.1.

- The compiler must output valid O'Caml object files that are accepted by the O'Caml linker.

- The operational semantics of the executable bytecode file running under the modified O'Caml virtual machine must match those detailed in the Acute Technical Report [13].
- The performance of the resulting bytecode must be comparable to that of the equivalent bytecode produced by the O'Caml bytecode compiler.

The shaded terms of the grammar in Figure 2.1 essentially represent the Acute version of the simply-typed lambda calculus, extended with marshalling operations. We subsequently extended the compiler to support the remainder of this grammar fragment during the third stage. Note that the grammar terms are given in their sugared form with full type annotations.

2.3 Development model

The spiral development model [5] was a clear choice for the software development. It specifies a form of evolutionary development using the waterfall development model [12] for every step. Each stage is evaluated before advancing on to the next iteration.

During the project it was possible to formulate a relatively complete, high-level specification from an early stage. This could be easily translated into a modular architecture for the compiler. The spiral development model allowed the sequential implementation of compiler modules and runtime extensions, but with the flexibility to iterate the development and evaluation of those that were more complex. For example, we were unsure of the exact details of the bytecode emission to the object files. Once development reached this latter stage, considerable time was spent prototyping and evaluating this module.

2.4 Languages

The bytecode compiler was written in the Fresh O'Caml programming language. O'Caml is a strongly typed, general purpose programming language from the ML family. Fresh O'Caml adds additional features to the language to allow the efficient manipulation of object-level syntax involving binding operations. Both the Acute compiler and runtime are written in Fresh O'Caml. The Fresh O'Caml language was ideally suited to the purpose of compiler implementation. For example, we found the fresh name generation especially useful. However, the use of this language also allowed us to easily interface to the Acute compiler in order to extract the intermediate language form.

During the second and third stages of the implementation, we used the C language to add marshalling facilities to the O'Caml runtime. The O'Caml developers have provided a well-documented C application programming

Standard library constants

$\mathtt{x}^n$

Types

$$\begin{array}{lcl} T & ::= & \mathtt{int} \mid \mathtt{string} \mid \mathtt{bool} \mid \mathtt{unit} \mid T \to T' \end{array}$$

Constructors

$$\begin{array}{lcl} C_0 & ::= & \ldots \end{array}$$

Operators

$$\begin{array}{lcl} op & ::= & (=_{int}) \mid (<) \mid (\leq) \mid (>) \mid (\geq) \mid (+) \mid (-) \mid (*) \mid (/) \mid - \end{array}$$

Expressions

$$\begin{array}{lcl}
e & ::= & C_0 \mid \mathtt{function}\ mtch \mid \mathtt{fun}\ mtch \mid op^n\ e_1 \ldots e_n \\
& \mid & x^n\ e_1 \ldots e_n \mid x \mid \mathtt{M}_M\mathtt{.x} \mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \\
& \mid & e_1\ \mathtt{;}\ e_2 \mid e_1\ e_2 \mid \mathtt{let}\ p\ \mathtt{=}\ e'\ \mathtt{in}\ e'' \\
& \mid & \mathtt{let}\ x\ \mathtt{:}\ T\ p_1 \ldots p_n\ \mathtt{=}\ e'\ \mathtt{in}\ e'' \\
& \mid & \mathtt{let\ rec}\ x\ \mathtt{:}\ T\ \mathtt{=}\ \mathtt{function}\ mtch\ \mathtt{in}\ e \\
& \mid & \mathtt{let\ rec}\ x\ \mathtt{:}\ T\ p_1 \ldots p_n\ \mathtt{=}\ e'\ \mathtt{in}\ e'' \\
& \mid & \mathtt{marshal}\ e_1\ e_2\ \mathtt{:}\ T \mid \mathtt{unmarshal}\ e\ \mathtt{as}\ T
\end{array}$$

Matches and patterns

$$\begin{array}{rcl}
mtch & ::= & p \to e \\
p & ::= & \mathtt{(}x\ \mathtt{:}\ T\mathtt{)} \mid \mathtt{(\_\ :}\ T\mathtt{)}
\end{array}$$

Signatures and structures

$$\begin{array}{lcl}
sig & ::= & \text{empty} \mid \mathtt{val}\ \mathtt{x}_x\ \mathtt{:}\ T\ sig \\
Sig & ::= & \mathtt{sig}\ sig\ \mathtt{end} \\
str & ::= & \text{empty} \mid \mathtt{let}\ \mathtt{x}_x\ \mathtt{:}\ T\ p_1 \ldots p_n\ \mathtt{=}\ e\ str \\
Str & ::= & \mathtt{struct}\ str\ \mathtt{end}
\end{array}$$

Source definitions and compilation units

$$\begin{array}{lcl}
sourcedefinition & ::= & \mathtt{module}\ \mathtt{M}_M\ \mathtt{:}\ Sig\ \mathtt{=}\ Str \mid \mathtt{mark\ MK} \\
compilationunit & ::= & \text{empty} \mid e \mid sourcedefinition\ \mathtt{;;}\ compilationunit \\
& \mid & \mathtt{includesource}\ sourcefilename\ \mathtt{;;}\ compilationunit
\end{array}$$

Figure 2.1: Supported source grammar

interface for such purposes. The O'Caml linker can be configured to output custom runtimes incorporating additional C object files. This code can then be easily invoked from within O'Caml bytecode.

The project made extensive use of testing throughout the implementation phases. A Perl script was created to automatically preprocess, compile and execute Acute test files. Perl is ideally suited to such file and text manipulation tasks. Benchmarking was conducted using a set of Bash shell scripts. The GNU Make system was used for high-level control during compilation, testing and benchmarking.

This dissertation was prepared in the Latex document preparation system. The facilities for the partitioning of source files and the typesetting of rules were found to be especially useful.

2.5 Starting point

Many areas of this project were either new or indeed experimental. The author's knowledge of compilers was limited to the details covered in the Part IB Compiler Construction [10] course. The Part II Types [11] course occurred towards the end of the Michaelmas Term and therefore it was necessary to independently study the lecture notes. Neither O'Caml nor Acute had been previously covered.

2.6 Code reuse

This project directly extended the Acute compiler, although the source trees were kept separate in order to facilitate updates to the Acute code. The import stage of the bytecode compiler invoked the Acute compiler in order to extract the intermediate language syntax tree.

We used the linking facility of the O'Caml compiler to create custom runtimes from our emitted bytecode object files. We directly invoked the linker from the command line.

It was necessary to study the high-level algorithms and data formats used by the O'Caml bytecode compiler during the bytecode generation and emission stages. Whilst we directly used both the O'Caml object file descriptor datatype as well as the instruction datatype, all other code was independently produced.

The test file format used by the automated regression testing system was based upon that used by Acute. However, the actual Perl testing program was independently produced.

2.7 Version control and backup

The GNU Revision Control System was used to track the changes between versions of files. Files were regularly checked into the system, especially before major changes were made. This allowed the easy roll-back of changes.

A shell script was used to make automated hourly snapshots of the source tree to the Pelican archive server throughout the project. The project work was also backed up onto Compact Disc format on a frequent basis. Although there was no need to restore from either facility during the course of the project, both were initially tested to ensure that recovery from disc failure was indeed possible.

2.8 Compilation techniques

Several useful references were found detailing compiler construction for functional languages [3], [4], [10]. Taken in conjunction with existing O'Caml design documents, this permitted the design of a small, moderately efficient compiler to target O'Caml bytecode. In this section we explore two key compilation techniques know as *closure conversion* and *hoisting*.

2.8.1 Closure conversion

In most lexically scoped, higher-order languages, a function is compiled into a closure. The closure consists of two parts: an environment and the code representation of a closed lambda abstraction. The dynamically created environment maps identifiers to values.

During compilation, lambda abstractions are converted into closures by a technique known as *closure conversion*. This separates program code from data. All terms are rewritten so that they are of closed form. This requires that any variables from the context must be explicitly passed as arguments to the function. We do this by creating an environment which is paired with the (now closed) code. Together, this forms a closure. The technique of closure conversion is very similar to *lambda lifting*, but instead makes the environment representation explicit.

Consider the source code in Figure 2.2. The function definition `f` is not of closed form; in addition to the formal parameter `x`, it also references the identifier `a`. Closure conversion transforms the code to make `a` an additional, explicit parameter of `f`.

We illustrate an untyped, source to source example of closure conversion in Figure 2.3 (based upon [9]). We explicitly pass `f` an environment containing the the required identifier bindings. The function body must now access the value of `a` by projecting it from the environment.

The environment in Figure 2.3 contains a redundant binding for `b`. It is certainly *safe* to pass in the values of all current identifiers in scope.

However, we can increase the runtime performance of the code by only passing in those bindings that are actually accessed. More formally, it is sufficient for the environment to only contain bindings for the free variables of `f`. An example of this optimisation is shown in Figure 2.4.

2.8.2 Hoisting

After closure conversion, nested (closed) lambda terms still remain. Using a *hoisting* operation, we name the closed lambda terms and lift them to the top level of the program. The nested lambda terms are replaced by the names of the hoisted closures. This completes the separation between code and data.

2.9 O'Caml bytecode

O'Caml bytecode is interpreted by the O'Caml virtual machine. The bytecode format is untyped. Garbage collection is used to reclaim unused memory. An O'Caml source file is compiled to a relocatable bytecode object file by using the `ocamlc` program. Once linked, a bytecode program can then be executed on the virtual machine by using the `ocamlrun` program.

Before designing the Acute bytecode compiler, it was first necessary to understand the operational semantics of the O'Caml virtual machine. The absence of official documentation did not facilitate this task. However, two partially relevant documents were found. The first sketched the basic bytecode level functionality of the O'Caml virtual machine [6]. Although the scope of the document was limited, it did provide an important introduction in the virtual machine operation. The second document outlined the early design of the ZINC virtual machine, which later evolved into the current O'Caml virtual machine [8].

The analysis of the bytecode semantics proceeded by using the opcode printing facility of `ocamlc`. This prints a sequence of bytecode mnemonics which correspond to the bytecode output. This allowed the author to understand the translation of relatively simple O'Caml programs. Towards the end of this stage it was possible to hand compile and run simple programs.

The second task was to understand the object file format. The object file contains a block of relocatable bytecode, in addition to a data structure detailing relocation information. One or more object files may then be linked together to produce a bytecode executable file. Again, no public documentation exists for the file format. Therefore it was necessary to manually analyse several hundred lines of the compiler source code in order to reverse-engineer the format.

2.9.1 Virtual machine operation

The O'Caml virtual machine uses a stack-based architecture incorporating an accumulator. Most operations implicitly operate upon the accumulator, taking additional arguments from the top of the stack as necessary. There are also operations to add and remove elements from the stack. For the purposes of this discussion, there are three important virtual machine registers:

`accu` The accumulator register holds the current working value. The combined use of an accumulator register and stack increases runtime performance.

`sp` The stack pointer register references the first free location on the heap-allocated stack. The O'Caml stack grows downwards towards lower memory addresses.

`pc` The program counter references the next instruction to execute.

In order to make the operation of the virtual machine more concrete, we shall consider the bytecode representation of the arithmetic expression `17 * 42`:

```
const 42
push
const 17
mulint
```

Upon execution, this code causes the value `42` to be placed into the accumulator. This value is then pushed onto the stack in order to allow the value `17` to be entered into the accumulator. The `mulint` instruction then destructively multiplies the two values, leaving the result in the accumulator. This operation is illustrated in Figure 2.5.

Referring to the fragment of O'Caml interpreter source code shown in Figure 2.6, we see that the `mulint` instruction causes the values held in the accumulator and the top stack position to be multiplied and placed into the accumulator. The post increment on `sp` removes the top stack value after multiplication. The `Val_long` and `Long_val` macros convert values between the interpreter and stack representations (see the next section). `Next` is a macro which increments the program counter register and jumps to the beginning of the enclosing switch statement.

Although the above example is relatively simple, it should give an indication of the operation of the O'Caml bytecode interpreter. In fact, the O'Caml bytecode compiler performs many optimisations to increase the common-case execution speed of the generated bytecode. For example, there is a complex system for the partial evaluation of functions.

```
let a = 17 in
let b = 42 in
let f = λx. x + a
```

Figure 2.2: Original code

```
let a = 17 in
let b = 42 in
let f = (λenv. λx. x + (env.a)) {a=a,b=b}
```

Figure 2.3: Naïve closure converted code

```
let a = 17 in
let b = 42 in
let f = (λenv. λx. x + (env.a)) {a=a}
```

Figure 2.4: Optimised closure converted code

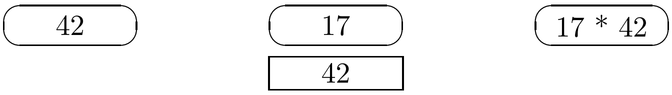


Figure 2.5: Virtual machine stack operation during execution

```
Instruct(MULINT):
  accu = Val_long(Long_val(accu) * Long_val(*sp++));
  Next;
```

Figure 2.6: Interpreter code for performing integer multiplication

2.9.2 Runtime data representation

The stack contains word-sized values. Each value is either an *immediate value* or a *block value*. A block value is a reference to a heap-allocated *block*. Any given integer value x is represented as the odd value $2 * x + 1$. In contrast, block references are word aligned and hence are always even values. This simple separation between the two kinds of data increases the efficiency of garbage collection.

2.9.3 Block structure

Each block has a *header*, followed by a *body* of zero or more contiguous words. The header structure is shown in the following diagram:

size	colour	tag
31 10	9 8	7 0

The `size` field indicates the total length of the block in words whilst the `colour` field is used by the garbage collector. The `tag` field indicates the kind of the block. For example, there is a closure kind.

2.9.4 Closures

An O'Caml function is implemented as a closure. The closure is constructed at runtime and is represented as a heap-allocated block. The first field of the body contains a pointer to the block of function bytecode. The following fields contain the environment values. More information about closure construction and manipulation is contained within Appendix F.

header	code pointer	env_0	$\cdots$	env_{n-1}

2.9.5 Object file format

An O'Caml object file contains a descriptor specifying relocation information for the contained block of bytecode. The high-level structure object file is summarised in the following diagram:

magic number
descriptor offset
relocatable bytecode
debugging information
descriptor

2.9.6 Modules

O'Caml modules permit the programmer to group related definitions under a single namespace. These definitions occur within a *structure*. The structure is bound to a name using the `module` keyword. A module has an associated *signature* that acts as an interface to the structure. This allows us to support abstraction through *encapsulation*.

Each O'Caml source file corresponds to a module definition, and may be separately compiled to produce an object file. For example, we may compile the source file `foo.ml` to produce the object file `foo.cmo`. This corresponds to a language-level module named `Foo`.

An evaluated module definition is stored as a block at runtime. The structure field values are sequentially stored within the body. This module block is then stored within a field of a single global block (`caml_global_data`). Fields from this global block are accessed using the `setglobal` and `getglobal` instructions.

During compilation, the locations of each module within the global block are unknown. Therefore we symbolically label fields within this block according to their module names, placing appropriate entries into the object relocation tables. All module dependencies are resolved during linking and hence each module definition can be allocated to a specific field within the global block.

Chapter 3

Implementation

This chapter documents the the project implementation. Due to space limitations, many of the low-level details are omitted; the main focus is upon the design choices and underlying algorithms. There are three parts, corresponding to the three phases of implementation, detailing (I) the core bytecode compiler, (II) support for marshalling and (III) support for modules and rebinding. For simplicity, we have included the details of module compilation within Part I.

Part I: The Bytecode Compiler

3.1 Overview

The high-level design of the Acute Bytecode Compiler (`abc`) is given in Figure 3.1. The O'Caml module system is used to partition the source files in a corresponding fashion. The design is standard for a compiler of a functional language [3], [4]. The bytecode generation phase follows the strategy used by the O'Caml bytecode compiler. A single 'pipeline' module passes data through the compilation stages, optionally pretty-printing the abstract syntax trees between each stage.

Although we describe the implementation of the finished compiler, it is important to note that many of the supported features were iteratively added. Initially, only a very restricted subset of Acute was supported. This allowed the sequential implementation of the compiler stages, with the main focus upon the correct architecture rather than on the specifics of Acute. The supported language could then be easily enlarged, finally including features such as recursive functions and modules.

Throughout this section, we shall consider the compilation of the simple Acute expression shown in Figure 3.2. This allows us to provide a concrete example of the bytecode compilation process.

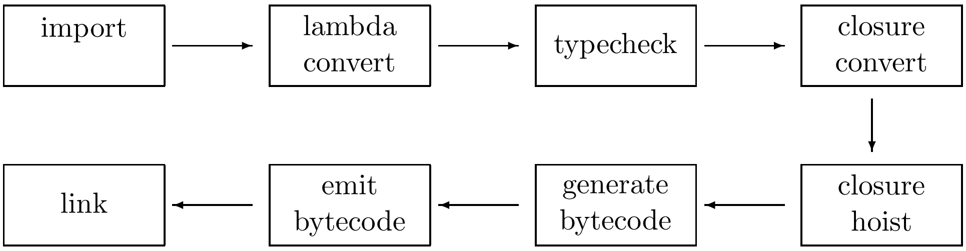


Figure 3.1: Compilation stages

```
let a = 0 in
let b = 1 in
let f = fun x -> x + a in
f 42
```

Figure 3.2: A simple Acute expression

3.2 Import

The purpose of this stage is to invoke the Acute compiler and translate the Acute intermediate language output into a format that can be easily manipulated by the subsequent stages. More concretely, the Acute output is converted into a simplified list of mark and module definitions, followed by a single expression. Each module definition contains a signature and structure. The signature is an association list of identifiers and types, whilst the structure is an association list of identifiers and values. Marks are represented as strings. The expression following the definitions is treated as a module containing a single identifier-value pair with an empty signature.

This stage also maps the Acute intermediate language structure values onto a simplified abstract syntax datatype. For those expressions that are supported, there is a one-to-one mapping onto the simplified datatype. Otherwise, an exception is raised detailing the unsupported expression. The signature types are treated similarly, with a straightforward mapping for the supported kinds.

The Acute intermediate language abstract syntax tree corresponding to our example is shown in Figure 3.3. Some of the nodes have been slightly simplified to aid comprehension. The dashed line denotes omitted nodes.

3.3 Lambda conversion

This stage converts the restricted Acute intermediate language expression of each structure field into an enriched version of the simply-typed lambda calculus. Many functional language compilers (including O'Caml) use a variant of the lambda calculus for their intermediate language. This representation is a clear choice for several reasons:

- The conversion to the lambda calculus from the Acute intermediate language is straightforward.
- Once closure converted, it is convenient to translate lambda calculus expressions into the target bytecode.
- The clear semantics of the lambda calculus facilitate optimising transformations if required.

The simply-typed lambda calculus requires that the formal parameters of lambda abstractions are assigned types. However, the Acute intermediate language is fully type-annotated. Therefore we naïvely use this information, verifying correctness during the typechecking stage.

Each identifier in the Acute intermediate language is freshly generated. Therefore we do not have to concern ourselves with issues of variable capture. The result of the lambda conversion of the example expression from the previous stage is shown in Figure 3.4. Again, some simplifications have been made with regard to the representation of constructors for the primitive values and operators. However, the shape of the tree is accurate.

The main complexity of the lambda conversion phase arises from the desugaring of the Acute `match` constructs. The associated `primmatch` nodes may have multiple child nodes to permit the deconstruction of variant types. This is redundant for our purposes. However, there is a straightforward mapping for the other expression kinds.

The algorithm for the translation of a `match` node is given in Figure 3.5. Essentially, a match creates a new identifier binding for the specified value. We therefore translate this into an application[1] of the value to a lambda abstraction. For example, we translate the root node of Figure 3.3 into an application of a lambda abstraction to the value `0`.

3.4 Typechecking

The typechecking stage imposes a well-formedness condition upon the lambda calculus expressions. Ideally this stage would be redundant; the Acute compiler also contains a typechecking stage, and therefore discards untypable

[1] We denote the application of a lambda abstraction by the symbol `@`

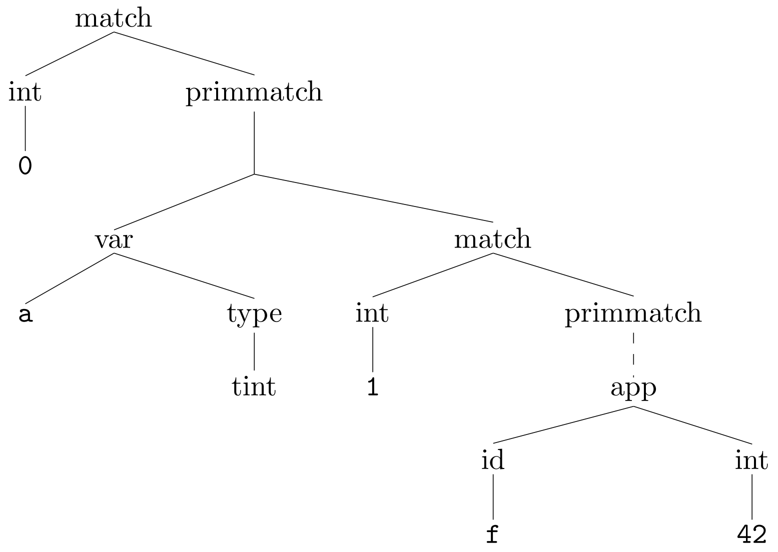


Figure 3.3: Acute intermediate language abstract syntax tree

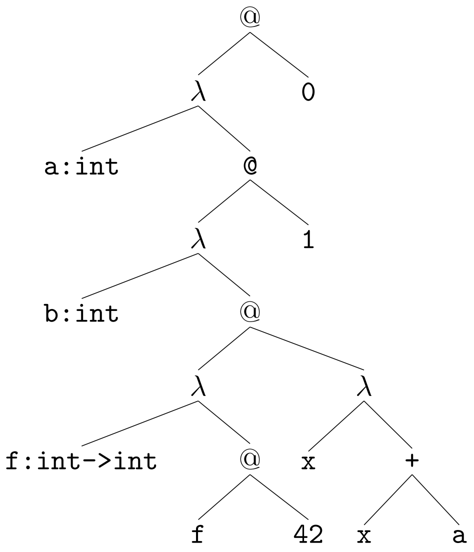


Figure 3.4: Extended lambda calculus abstract syntax tree

```
let rec conv_prim_expr e =
  match e with
  | Astsimple.Match (x,y) ->
      Astlambda.App(conv_prim_mtch y, conv_prim_expr x)
  | ...
```

Figure 3.5: Code to translate a match node

programs. However, this stage provides a valuable correctness confirmation in the presence of possible compiler bugs (although none were found).

Typechecking occurs at the module level. In outline, the algorithm examines each structure field, calculates the type and verifies that this matches that stored in the associated signature field. Failure of typechecking causes a diagnostic message to be printed.

The core typing algorithm is standard for the simply-typed lambda calculus. However, based upon the typing algorithm found in the Acute Technical Report [13], it was extended to support modules. The full set of typing rules is given in Appendix B.

The main rule for typing a structure is given below. E maps module identifiers to signatures whilst Γ maps local identifiers to values. Where Γ is omitted we implicitly assume that this environment is empty. The linear scoping of structure fields is clearly illustrated; once a field has been typed, it is added to the list of assumptions for the typing of subsequent structure fields.

$$\frac{\mathrm{E},\ \Gamma, x : T \ \vdash\ \mathit{str}\ :\ \mathit{sig} \qquad \mathrm{E},\ \Gamma\ \vdash\ v\ :\ T}{\mathrm{E},\ \Gamma\ :\ \textbf{let}\ \mathrm{x}_x\ =\ v\ \mathit{str}\ :\ \textbf{val}\ \mathrm{x}_x\ :\ T\ \mathit{sig}}$$

Once all structure fields have been successfully typechecked, the signature is added as a typing assumption for the use of latter modules. Again, we see the linear scope in terms of module definitions. The rule for typing a module is given below:

$$\frac{\begin{array}{c}\mathrm{E}\ \vdash\ \textbf{module}\ \mathrm{M}_M\ :\mathit{Sig}\ =\ \mathit{Str}\ :\ \mathrm{M}_M : \mathit{Sig} \\ \mathrm{E}, \mathrm{M}_M : \mathit{Sig}\ \vdash\ \mathit{defs}\ :\ \textbf{ok}\end{array}}{\mathrm{E}\ \vdash\ \textbf{module}\ \mathrm{M}_M\ :\mathit{Sig}\ =\ \mathit{Str}\ \mathit{defs}\ :\ \textbf{ok}}$$

The accumulated typing assumptions are used when typing an expression that includes a field projection from an earlier module:

$$\frac{\mathrm{M}_M \in \mathrm{dom}(E) \qquad \mathrm{E}(\mathrm{M}_M)\ =\ \textbf{sig}\ \mathit{sig}\ \textbf{val}\ \mathrm{x}_x : T\ \mathit{sig'}\ \textbf{end}}{\mathrm{E},\ \Gamma\ \vdash\ \mathrm{M}_M.\mathrm{x}\ :\ T}$$

3.5 Closure conversion and hoisting

The closure conversion stage converts the lambda calculus expression of each structure field into a set of one or more hoisted closures. We simultaneously perform hoisting in this stage because it adds little complexity to the closure conversion algorithm.

3.5.1 Closure data structure

Recalling the discussion of closures from the preparation section, each closure consists of a data structure containing the runtime environment paired with a block of function code. The compile-time representation of a closure consists of a record containing the following fields:

Closure name This is represented by a freshly generated identifier. Although this identifier does not appear in the resulting bytecode, it allows us to easily reference closures.

Function name This represents the bound name of the function by which the closure code may perform a recursive call.

Formal parameter This corresponds to the formal parameter of the function code.

Environment This represents the formal parameters of the closure environment. When outputting code to instantiate a closure, these parameters tell us which values should be passed to the closure environment.

Code This is identical to the simply typed lambda calculus datatype from the previous stage, with the exception that lambda abstractions are replaced by the name and actual environment parameters of the corresponding closure.

For example, the top-level closure of a structure field is called `cmain`. There is a single fresh formal parameter `mainx`. The environment is necessarily empty. The code constructs the closure for `c1` (also with an empty environment) and applies it to the value `0`. We adopt the following representation of this closure during pretty printing:

```
Closure (cmain, mainx, [])
  (c1 : []) @ 0
```

3.5.2 Closure conversion algorithm

The main closure conversion function performs a pattern match on the input lambda expression. Figure 3.6 highlights the part of the algorithm that converts a lambda abstraction. We see that a fresh name for the closure is constructed on lines three and four. On lines five to ten a closure is recursively constructed and concatenated onto the list of existing closures. The closure name and environment parameters are then returned.

We use the free variable optimisation of Section 2.8.1 on line nine. We see that the environment is restricted to contain only those variables that are free in the lambda abstraction. This optimisation was introduced to limit the size of the closure environments; initially the environment contained all of the identifiers currently in scope.

The closure conversion algorithm also accepts a list of previously converted module definitions. This allows us to convert references to external module fields into offsets within modules. For example, suppose the field `x` is the first (or zeroth) within a module `M`. The identifier `M.x` would then be translated to the pair `(M, 0)`. This conversion is performed relatively late in order to abstract away from implementation complexities. However, it is required for the subsequent bytecode conversion stage.

Continuing with the compilation of our example expression, we obtain the set of closures highlighted in Figure 3.7. We see that closure `c3`, corresponding to the body of our original function `f`, contains the environment variable `a`. The separation between code and data is now explicit.

3.6 Bytecode generation

The bytecode generation stage converts a list of closures into into a single list of 'high-level' instructions. Each of the instructions has an immediate, one-to-one correspondence to an actual O'Caml bytecode instruction, but in a form that is amenable to manipulation. For example, we label instructions by their corresponding closure name instead of referencing them by a relative offset.

3.6.1 Overview

The algorithm to generate bytecode for a module proceeds in three distinct phases. Initially we generate a block of 'top-level' bytecode. During this stage we maintain a list `cs` of closures for later conversion to bytecode. In outline, the algorithm iterates over each structure identifier within a module. For each of the associated closure lists, we perform the following operations:

1. Generate 'top' bytecode for the expression contained within the `cmain` closure.

2. Add the remainder of the associated closures to `cs`.

Ignoring the untranslated closures in `cs`, the generated code will calculate each field value in turn, pushing the result onto the virtual machine stack.

```
let rec conv_lambda (ds:def list) (e:lambda) =
| Astlambda.Fun (x,t,y) ->
      let n = Ident.fresh_prettyname
        ("c" ^ (string_of_int (incr count;!count)))
      in let c = { name  = n;
                   arg   = (x,t);
                   fnrec = None;
                   code  = conv_lambda ds y;
                   env   = IdentSet.elements (fv e)}
      in closures := c :: !closures;
        Astclosure.ClosureNonRecName (n, t,
          List.map (fun x -> Astclosure.Id(x)) c.env)
| ...
```

Figure 3.6: Closure conversion code for a lambda abstraction

```
Closure (cmain, mainx, [])
  (c1 : []) @ 0
Closure (c1, a, [])
  (c2 : [a]) @ 1
Closure (c2, b, [a])
  (c4 : []) @ (c3 : [a])
Closure (c4, f, [])
  f @ 42
Closure (c3, x, [a])
  (+, x, a)
```

Figure 3.7: Hoisted closures

This order is essential because later fields may make use of the values of earlier fields. However, this means that the first field value becomes the last (deepest) stack element. We wish to create a block where the first block field corresponds to the first structure field. Therefore we must reverse the order of the stack elements before creating the block of module values. We therefore generate bytecode to perform the following:

1. Reverse the order of the stack values.
2. Create a block in the accumulator from the reversed stack values.
3. Pop the remaining stack elements from the stack.
4. Store the block in the global block.

Returning to `cs`, should it be non-empty then the block of top-level bytecode will contain references to untranslated closures. We therefore translate the remaining closures in `cs` according to the following algorithm:

1. Label the first instruction of the block of top-level bytecode as `main`.
2. Translate the remaining closures within `cs`, adding the new instructions to the head of the top-level bytecode.
3. Add a branch to `main` from start of the bytecode.

This strategy is consistent with that employed by the O'Caml compiler. During linking, the blocks of bytecode are sequentially concatenated. Therefore we require that the block of top-level bytecode occurs last so that execution can continue onto subsequent bytecode.

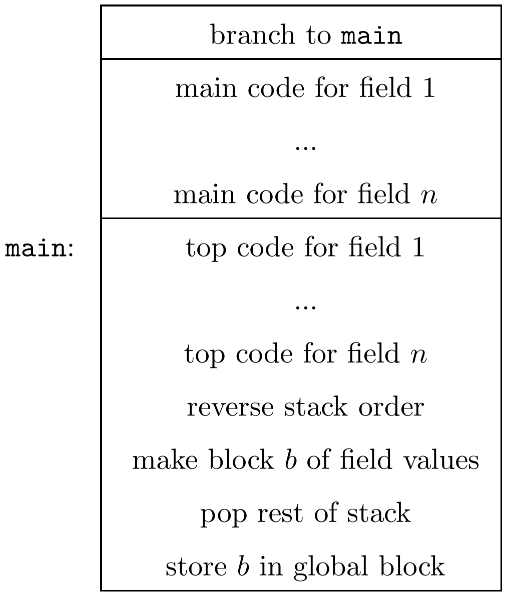
The high-level structure of the resulting bytecode is shown in Figure 3.8. The list of instructions is now in a format suitable for conversion to actual O'Caml bytecode instructions and subsequent emission to an object file.

3.6.2 Translation of a closure

The above discussion omits the important details of how to translate a closure to a block of high-level bytecode. The main complexities arise out of having to track the size of the current stack frame and the module identifiers that have already been translated.

The closure translation function accepts the following arguments in addition to the closure:

Identifier list This contains the previously translated structure field identifiers for the current module. The main closure for a field may directly reference previous field values. In this sense, it is not strictly closed. Should the top-level closure need to access these, then the list gives the order and hence the position on the stack at which the field value may be located in the current, top-level stack frame.



branch to `main`

main code for field 1

...

main code for field n

`main`: top code for field 1

...

top code for field n

reverse stack order

make block b of field values

pop rest of stack

store b in global block

Figure 3.8: High-level structure of object code

Bytecode instruction list This contains the current bytecode instructions. Whilst generating bytecode, we traverse the abstract syntax tree in postorder, passing the current list of bytecode instructions as an additional argument to the expression translation function. This means that all future instructions (in terms of execution order) are available, facilitating certain peephole optimisations.

Each closure contains a single expression which we recursively translate. Whilst translating this expression we must keep track of the number of values which have pushed onto the current stack frame so that we can correctly access identifier values. When we encounter an identifier, it can be one of three kinds:

Closure argument When a closure is applied, the argument is placed as the first element upon the new stack frame. We can therefore access it at as the bottom element of the current stack frame.

Environment field identifier We look up the identifier within the containing closure, and generate an instruction to access it from the correct position within the current environment.

Module field identifier This can only occur within the main closure for a field. The list of translated identifiers reveals the location within the stack.

```
let conv_expr : closure_body -> closure -> ident list
                 -> int -> instr list -> instr list
  = fun expr clos ids sz cont ->
      match expr with
      | Id x ->
          locate_id x clos ids sz :: cont
      | Const x ->
          conv_const x :: cont
      | Op (x,ys) ->
          conv_expr_list ys clos ids sz
            (conv_op_or_econst x cont)
      | ClosureNonRecName (x,t,args) ->
          conv_args args clos ids sz
            (Pclosure ((List.length args), x) :: cont)
      | ...
```

Figure 3.9: Closure expression translation

A fragment of the main algorithm for the translation of a closure expression is given in Figure 3.9.

The results of the translation of the example expression are shown in Figure 3.10. We can clearly see the correspondence to the previous list of closures of Figure 3.7.

3.7 Bytecode emission

This stage is responsible for translating each of the instructions in a sequence of high-level bytecode into the corresponding O'Caml bytecode instructions. We maintain a string buffer to which we write the instruction values. This block of relocatable bytecode is then written into an object file. The algorithm has two main complexities.

- Each reference to a labelled instruction must now be translated into a relative bytecode offset. For example, the instruction to construct a closure requires the relative word-offset of the associated function code.

- A relocation table must be constructed. It contains the bytecode location references to external resources as well as values such as string constants that are too large to be directly included in the bytecode.

When we encounter a label *reference*, we invoke the `ret_label` function with the label name. Should the label have already been defined, then the

```
          branch main
c1:       int 1
          push
          acc 1
          closure c2, 1
          apply 1
          return 1
c2:       envacc 1
          closure c3, 1
          push
          closure c4, 0
          apply 1
          return 1
c4:       int 42
          push
          acc 1
          apply 1
          return 1
c3:       envacc 1
          push
          acc 1
          add
          return 1
main:     int 0
          push
          closure c1, 0
          apply 1
          push
          acc 0
          makeblock 1, 0
          pop 1
          setglobal Let!
```

Figure 3.10: High-level bytecode

offset from the current string buffer output position is returned. Otherwise, the offset of reference is associated with the label and the value of zero is returned.

In the case of a label *definition*, we invoke the `add_label` function with the label name and offset of definition. If the label has been previously referenced, then we backpatch the referencing locations with the actual offset. We also record the location of the definition within the association list for later references.

If we encounter a reference to one of the following kinds, we output a temporary offset of zero and add an entry into the relocation table recording the value and position of reference. The offsets are patched during linking when the location of the external resources become known.

- String constant
- C primitive
- Global block

A fragment of the core translation algorithm for a high-level instruction is shown in Figure 3.11.

For each instruction that we write to the string buffer, we first check that there is sufficient space. If not, we double the size of the buffer and copy across the current buffer contents. We initially set the buffer size to 1024 bytes, although this could be increased if larger programs were routinely compiled.

Once all of the bytecode has been translated, we open an output stream to an appropriately named file, and marshal the bytecode and relocation table.

3.8 Object linking

We use the dual-function O'Caml bytecode compiler `ocamlc` to generate a custom runtime containing the linked bytecode. We pass the following options to the linker:

Executable name The name of the emitted bytecode executable file.

O'Caml libraries The name of the libraries containing the O'Caml functions referenced by the bytecode. These are required when we interface to existing O'Caml code.

O'Caml object files The correctly ordered list of object files that we have just emitted.

```
let conv_prim p =
  match p with
  | Preturn n ->
      output_op opRETURN;
      output_int n
  | Plabel l ->
      add_label l
  | Penvacc n ->
      output_op opENVACC;
      output_int n
  | Pacc n ->
      if n = 0 then output_op opACC0
      else (output_op opACC; output_int n)
  | Papply n ->
      if n = 1 then output_op opAPPLY1
      else if n = 2 then output_op opAPPLY2
      else if n = 3 then output_op opAPPLY3
      else raise (UnsupportedOp "Papply, n > 3")
  | Pclosure (nv,l) ->
      output_op opCLOSURE;
      output_int nv;
      output_int (ret_label l)
  | ...
```

Figure 3.11: Algorithm for the translation of a high-level instruction

C object files The name of the C object files containing the user-defined primitive functions referenced by the bytecode. These are required when we extend the runtime to permit marshalling.

3.9 Interfacing to existing O'Caml modules

We extended the project to allow the generated bytecode to access O'Caml modules. This permitted compiled Acute code to make use of the O'Caml graphics library, for example. Note that this facility is already supported by the existing Acute interpreter; the Acute abstract syntax type contains a constructor for O'Caml module field references. However we were initially unsure how we might incorporate this functionality into our generated bytecode.

In principle, it is relatively easy to invoke other O'Caml bytecode. We can use the `getglobal` instruction to retrieve the block of module structure values. Providing that we know the relative position of the structure field, we can then access the required field within the block. We therefore performed the following modifications to the bytecode compiler:

Typechecker In order to typecheck references to the O'Caml module fields, each must be assigned a type. Acute has tool support to automatically extract these types from O'Caml source files. However, given the limited number of O'Caml functions that were supported, it was relatively easy to use a hard-coded mapping.

Bytecode generation We have to convert the module field names to concrete offsets within modules. Again, a hard-coded mapping was used.

Object file linking The O'Caml object files containing the referenced code must be passed to the linker.

In each case, an O'Caml 'shim' module was required to wrap calls to the intended target module. The shim modules provided simplified interfaces, only using those types which were supported by `abc`. Shims for the graphics module and input/output functions already existed in the Acute code for this purpose. However, an additional interface to the TCP module was implemented.

Part II: Marshalling

3.10 Overview

The second stage of the implementation involved extending the `abc` compiler and the O'Caml runtime to support the marshalling of program values. Initially, we consider marshalling in the absence of module references.

We return to our successor function `succ`. This is represented at runtime by a closure. Marshalling necessitates that we convert this block value into a linear sequence of bytes. This marshal string must specify the type of the marshalled value to ensure type safety at unmarshal time. Furthermore, it must also contain a representation of the *reachable* bytecode instructions. Finally, each of the closure environment values must also be marshalled. In effect, we are capturing the complete runtime environment of `succ`. Note that the marshal mark only becomes relevant when we consider the marshalling of module references.

```
let succ n = n + 1 in
let s = marshal "MK" succ in
IO.send s
```

The unmarshalling operation converts the received byte-string back into a closure value. The type of the marshalled value must be compared to the expected type, with an exception raised if they differ. The code and environment values may then be unmarshalled. This requires the dynamic allocation of code. Note that the bytecode compiler generates relocatable bytecode, hence facilitating this operation. Execution can continue once the closure has been reconstructed.

```
let s = IO.receive () in
let succ = unmarshal s as (int -> int) in
succ 42
```

3.11 Design

3.11.1 Primitives

The transfer of function values between runtimes necessitates the manipulation of the virtual machine state; unmarshalling a function value requires the dynamic allocation of closure code, whilst rebinding is dependent upon maintaining the dynamic mark/module structure.

We follow the recommended method of extending the virtual machine by generating a custom runtime. This operation is performed by the linker; we simply invoke it with the C object files containing the marshalling primitives.

The resulting executable file is fully self-contained. Note that all runtimes participating in marshalling operations must be similarly extended.

The use of C primitives means that few modifications need to be performed to the O'Caml bytecode generation stage. We marshal a value by simply invoking the marshalling primitive from the bytecode, minimising our reliance upon the untyped bytecode instructions. The primitive then performs the necessary computation before returning a single value into the accumulator.

Note that we originally proposed using dynamic linking of C libraries. Although this method is indeed feasible, the custom runtime generation is performed entirely by the existing O'Caml linker. This reduced the amount of implementation work.

3.11.2 Types

We must perform a runtime check when unmarshalling a value v to verify that the expected and actual types of v are equal. This requires that we include a representation of the type of v within the marshal string. We use a pretty-print of the type for this purpose. This aids debugging and has the advantage that testing for type equality becomes a simple string comparison.

It is important to note that the method outlined above only guarantees type safety if the marshal strings are unaltered. For increased robustness, we might include a hash of the marshalled value and type. In an untrusted environment we might wish to explore signing the marshal strings.

3.11.3 Wire format

We require the easy conversion between values and the wire format for the marshal string. A self-delimiting structure is used to assist parsing, with all variable length fields prefixed by the field length. Fields are tagged to differentiate between immediate and closure values. The full grammar is given in Appendix C. It is recommended that the marshalling details are read in conjunction with this grammar.

3.12 Marshalling

The marshalling primitive converts a value to a linear sequence of bytes containing the value type and the marshalled value. We give a diagrammatic representation of this format:

type	value

The primitive maintains an internal character buffer to which values are written. Once marshalling is complete, the buffer contents are copied into

```
string "int -> int"
push
string "MK"
push
closure c1, 0
prim marshal, 3
```

Figure 3.12: Bytecode for marshalling the successor function.

a block on the O'Caml heap. A reference to this marshal string is then returned.

We initially discuss the interface to the marshalling primitive, before exploring the specifics of code reachability and value marshalling.

3.12.1 Interface

We invoke the marshalling primitive with the marshal mark `mk`, value `v` and type string `t`. All expressions are monomorphically typed, hence allowing us to statically calculate `t` at compile-time. The C interface is given below:

```
value marshal (value mk, value v, value t);
```

We illustrate the invocation of this primitive in Figure 3.12. Our successor function (compiled to closure `c1`) is marshalled with respect to the `MK` mark. After the execution of the call to the primitive, the marshalled function will be left in the accumulator for use in subsequent computation.

3.12.2 Code reachability

When marshalling a closure, we must include all instructions that are reachable from the closure code pointer *cp*. We perform an *abstract interpretation* of the referenced instructions to calculate a safe approximation to the runtime code reachability. In effect, we implement a *very* restricted O'Caml bytecode interpreter; we are only concerned with the limits of the program counter value during any possible future executions of the closure code.

In outline, we iterate over each bytecode instruction. If we encounter an instruction to construct a closure, then we recursively follow the specified code offset. We always assume that all such closures are constructed; undecidability prevents us from calculating the exact runtime reachability. Encountering a return instruction causes our algorithm to return the current limits of the emulated program counter value. The core of this algorithm is given in Figure 3.13.

```
(code *, code *) limits (code *cp)
{
  code *low  = cp;
  code *high = cp;

  // iterate over each closure instruction
  for (; *cp != Return; cp++)
  {
    switch (*cp)
    {
      // closure construction
      case Closure(_, cp'):
        code *low', *high';

        // calculate reachability
        (low', high') = limits(cp');

        // keep maximal sequence
        low  = MIN(low, low');
        high = MAX(high, high');
        break;
    }
  }

  // return reachability limits
  return (low, MAX(high, cp));
}
```

Figure 3.13: Pseudo-code for calculating code reachability

We give a formal definition of code reachability in Appendix D. It is recommended that this section is consulted for a more detailed account of this complex operation.

3.12.3 Immediate values

In the case where v is an immediate value, we directly copy each of the byte values into the character buffer.

3.12.4 Closure values

If v is a closure value then we must marshal all bytecode instruction that are reachable from the code pointer, as well as each of the environment values.

Code

We marshal the closure code by first calculating code reachability from the code pointer. All code that is inclusively contained between the returned pair of limits is directly copied into the character buffer. The wire format also contains a code offset which points to the instruction originally referenced by the closure code pointer. This information permits the reconstruction of the closure.

Note that the O'Caml virtual machine performs a 'threaded code' optimisation at runtime whereby opcodes are converted into offsets into a lookup table. In order to simplify code marshalling we disable this optimisation. This is achieved by trivially modifying two of the O'Caml compiler header files and recompiling O'Caml. The benchmarks of Section 4.6 demonstrate that this modification has a minimal impact upon runtime performance.

Environment

An environment value may be of an immediate or closure kind. In each case, we recursively apply the marshalling algorithm. The wire format specifies the number of environment fields.

3.13 Unmarshalling

The unmarshalling primitive iterates through the characters of a marshal string *s*. The type of the marshalled value is compared with the expected type *t*. An exception is raised if the two differ. Otherwise the original marshalled value is reconstructed. Again, we briefly discuss the interface before exploring the specifics of unmarshalling immediate and closure values.

3.13.1 Interface

The interface to the unmarshalling primitive accepts two string values *s* and *t*, which correspond to the marshalled value and the expected type respectively:

```
value unmarshal (value s, value t);
```

3.13.2 Immediate values

In order to unmarshal an immediate value, we directly return the byte values that we have received in the marshal string.

3.13.3 Closure values

We unmarshal a closure by first allocating a closure block on the O'Caml heap. The code pointer and environment fields of this block are filled during the subsequent unmarshalling operation. Once complete, we return a reference to this block.

Code

The marshalled closure code is copied into a block of memory allocated upon the C heap. We then set the closure code pointer to the correct offset within this memory. Note that the use of the C heap for code allocation simplifies the implementation; we do not have to concern ourselves with the operation of the O'Caml garbage collector. However, this does mean that the memory is only freed when the runtime terminates.

Environment

Each environment field is recursively unmarshalled and then stored within the correct field of the closure block.

Part III: Modules and Rebinding

3.14 Overview

The final implementation stage involved extending the bytecode compiler and runtime to support modules and rebinding. We have already covered many the aspects of module compilation in Part I. Therefore in this section we mainly focus upon the enhancements to the runtime extensions.

Returning to our second example from the introduction, our function `f` references modules `M1` and `M2`. These module references occur through the use of `getglobal` instructions in the closure bytecode. We might attempt to marshal the closure code using the same strategy as that employed during the second phase. However, upon unmarshalling in a second runtime, these `getglobal` instructions would most probably point to the incorrect modules in the new global block, leading to unpredictable behaviour. *The execution environment of f would have changed.*

```
module M1 : sig val x:int end = struct let x = 17 end
mark "MK"
module M2 : sig val y:int end = struct let y = 42 end
let f () = M1.x + M2.y in
let s = marshal "MK" f in
IO.send s
```

The correct approach to this problem, consistent with the semantics of Acute, is to either rebind such references to the new module locations at unmarshal time, or else marshal the referenced modules along with the value. In the case of `f`, we rebind the reference to `M1.x` because the containing module occurs prior to the marshal mark `MK`. In contrast, we marshal module `M2` along with `f` because this second module is located after the mark.

Upon unmarshalling the byte-string, we must now dynamically allocate the marshalled module `M2`. In most cases, the position of this module in the new global data block will have changed from its original location. During the unmarshalling of `f`, we therefore scan the closure code, patching the `getglobal` references to `M1` and `M2`.

```
module M1 : sig val x:int end = struct let x = 17 end
let s = IO.receive () in
let f = unmarshal s as (unit -> int) in
f ()
```

Conveniently, the O'Caml virtual machine already performs redex-time instantiation of module fields: module references that are contained within a closure are only actually evaluated during an application. This considerably simplifies the implementation of rebinding.

3.15 Marshalling

We extend the marshalling algorithm outlined in Part II. Closure code may now reference arbitrary modules. We first outline the algorithm used to calculate module reachability, before addressing the issues of module marshalling and relocation.

The interface to the marshalling primitives remain unchanged. However, our marshal string now additionally contains the marshal mark, a relocation table *reloc* and marshalled module definitions mod_i.

type	mark	reloc	mod_0	$\cdots$	mod_{n-1}	value

3.15.1 Initialisation

The compiler of Part I generates bytecode executables that are devoid of the original Acute mark/module structure. However, such structure is necessary in order for the marshalling primitives to be able to determine the relative positions of modules with respect to marshal marks. Furthermore, this information must be dynamically updated as additional modules are unmarshalled.

We know at *compile-time* which modules and marks will initially be present. This information must be passed to the runtime when the bytecode is executed. We therefore extend `abc` to output initialisation code. The code reconstructs the compile-time mark/module structure as a linked-list of blocks, with each module name mapping to a field position (*global number*) in the global block. We invoke an initialisation primitive with the block structure. This primitive constructs an in-memory *definitions table* for use by the marshalling primitives.

Returning to our example code, we shall assume that modules `M1` and `M2` are located at global numbers `17` and `18` respectively. The initialisation code therefore constructs the following definitions table at runtime:

```
module M1 at offset 17
mark MK
module M2 at offset 18
```

3.15.2 Module reachability

When marshalling a closure value it is necessary to determine the referenced modules. Although this information is not explicitly contained within our program, we can again perform an abstract interpretation of the reachable bytecode to determine such *module reachability*.

In outline, our algorithm iterates over each instruction that is reachable from the code pointer. If we encounter a `getglobal` instruction, then we add the referenced global number to the list of reachable modules. We then retrieve the module block from the runtime and recursively iterate over the code that is reachable from each of the module field values. Again, undecidability prevents us from obtaining accurate runtime module reachability information. We therefore assume that all reachable `getglobal` instructions are executed. This gives us a safe approximation. The core algorithm is outlined in 3.14.

Note that module reachability progresses through static code and runtime module values. We give a formal definition of module reachability in Appendix E. We recommend that this section is consulted for a more detailed account of this complex operation.

Modules `M1` and `M2` are reachable from the closure code of our example function `f`.

3.15.3 Relocation table

We construct a *relocation table* for the reachable modules. This is used during the unmarshalling process to adjust module references. We create this table by pruning the entries for unreachable modules from the definitions table. Additionally, we omit those marks which occur prior to the marshal mark. We encode the resulting table and copy it into the marshal string.

The relocation table for our example function `f` is identical to the definitions table; `M1` and `M2` are both reachable from the closure code corresponding to `f`.

3.15.4 Module encoding

Once we have calculated the set of reachable modules, we use the definitions table to partition these modules with respect to the given marshal mark. We only marshal those modules that occur after the mark.

We marshal a module by iterating over each field, applying the standard value marshalling algorithm; each module field is either of an immediate or closure kind.

Only `M2` is marshalled along with our example function `f`. `M1` occurs prior to the marshal mark `MK` and therefore will be rebound upon unmarshalling.

3.16 Unmarshalling

During the unmarshalling of a value, we first decode the relocation table. We then dynamically allocate the included modules. We use the relocation table and marshal mark to patch any global numbers in the unmarshalled module and value code. This ensures that they correctly reference any modules in the new runtime, hence correctly restoring the execution environment.

```
(module set) mod_reach (code *cp)
{
  module set ms = {};

  // iterate over each closure instruction
  for (; *cp != Return; cp++)
  {
    switch (*cp)
    {
      // closure construction
      case Closure(_, cp’):
        ms = mod_reach(cp’) ∪ ms;
        break;

      // module access
      case Getglobal(n):
        // add n to reachable modules
        ms = {n} ∪ ms;

        // retrieve runtime module definition
        mod_def m = get_global(n);

        // iterate over each field value
        foreach (field f in m)
        {
          // calculate reachability
          if (closure(f))
          {
            ms = mod_reach(code(f)) ∪ ms;
          }
        }
        break;
    }
  }

  // reachable modules
  return ms;
}
```

Figure 3.14: Pseudo-code for calculating module reachability

3.16.1 Relocation table

The relocation table is decoded and stored within a temporary data structure. We reference this data during code patching.

3.16.2 Code patching

We patch the unmarshalled code to reference the correct modules by iterating over each reachable instruction. Upon locating a `getglobal` n instruction we consult the relocation table to map to the module name. This enables us to locate the correspondingly named module in the current runtime, and hence map to the new module number n'. We patch n to n'.

3.16.3 Modules

Each module definition is sequentially extracted from the marshal string. The module fields are stored within a newly-allocated block. This block is then stored within the next free field in the global block. Although this may require increasing the size of the global block, the virtual machine API provides an unpublished function for this purpose.

Each of the newly-allocated module fields may be of a closure or immediate kind. In the case of a closure kind, the reachable code is patched.

The new module definition must now be entered into the current runtime's definitions table. We extract the module name from the relocation table, and store the name and new global number as the last entry in the definitions table. Additionally, we add any marshalled marks that immediately follow this definition.

Returning to our example, we shall assume that `M1` has a global number of `42` and that the next free location in the global data block is `43`. After unmarshalling `M2` we obtain the following definitions table:

```
module M1 at offset 42
module M2 at offset 43
```

3.16.4 Closure values

A closure value is unmarshalled as before, with the additional requirement that all reachable bytecode must be patched. This correctly restores the execution environment.

The result of patching our newly unmarshalled closure is that `M1.x` is rebound to the existing module `M1`. Reference `M2.y` is updated to point to the new location of the unmarshalled module `M2`.

Chapter 4

Evaluation

4.1 Overview

In this section we examine both the project process as well as validating the requirements in relation to the delivered software. Additionally, we demonstrate the use of automated regression testing for the rapid evaluation of software functionality. We then analyse the performance benefits offered by compiling the Acute language into O'Caml bytecode. Finally, we explore a more ambitious example to demonstrate that the implemented compiler and runtime extensions support a relatively expressive fragment of Acute.

4.2 Code analysis

A total of 5250 lines of code were produced. We give a brief summary in the following table:

Language	*Purpose*	*Lines of code*
Fresh O'Caml	Bytecode compiler	2700
C	Marshalling primitives	1200
Acute	Test cases	900
Make	Makefiles	200
Perl	Test utility	100
Shell script	Benchmark framework	100
Gnuplot script	Benchmark graphs	50

4.3 Scheduling

A large amount of detailed planning and design occurred during the first phase of this project. This permitted the planned schedule to be followed for the entirety of the implementation phase of the project. Furthermore, the

Christmas vacation ‘buffer time’ was used to begin the dissertation writeup and bring forward the implementation of the module extensions.

4.4 Validation of requirements

The implemented software satisfied all of the original requirements. The compiler was capable of accepting the complete Acute fragment given in Figure 2.1, including recursive functions and simple modules. This greatly exceeded the marshalling-enabled lambda calculus fragment that was required by the acceptance criterion. The emitted object files were successfully linked by `ocamlc`. They could then be executed by the customised O’Caml virtual machine. This permitted the dynamic transfer of function code between runtimes.

The agreement between the operational semantics of the original Acute language and the interpreted bytecode was experimentally verified for the shaded grammar of Figure 2.1. However, there were some subtle differences in the semantics for rebinding. Specifically, we chose to ignore all module versioning information; the Acute semantics specify that rebinding can only occur to identical modules by default. Furthermore, we rebound references to a module field at unmarshal time instead of at instantiation time. Rebinding at instantiation time would require further extensions to the virtual machine to introduce a level of indirection for module field accesses.

An additional complication arose from marshalled bytecode containing references to external resources. Whilst we correctly handled modules, it is also possible for bytecode to access block-allocated string constants. We detected these rare occurrences during the calculation of code reachability and raised an error. Although we could have extended the primitives to correctly handle such references had time permitted, arbitrary O’Caml bytecode may also access other block-allocated constants as well as C functions. We believe that it would be easier to marshal such code if relocation information was retained at runtime.

Although these differences are appreciable, we have nonetheless achieved our goal of demonstrating a feasible low-level implementation of the marshalling primitives. Furthermore, the performance of the emitted bytecode files and modified runtime was comparable to that of `ocamlc` produced code running on the unmodified O’Caml virtual machine. We explore this area in a subsequent section.

4.5 Testing

Throughout the course of the compiler implementation, each stage of the compiler was tested as early and completely as possible. The rapid feedback offered by the spiral development process was a great advantage; by the time that the last compiler module was implemented, we were confident that all of the basic language features were supported. This allowed a quick progression to the implementation of the more ambitious features.

The main source of complexity during the project was the implementation of the syntax tree manipulations and the marshalling primitives. In order to test the translations, a set of comprehensive test cases were implemented for each syntactic component of the supported Acute source language. Combined with the pretty-printing facilities, this permitted each module to be tested as it was added into the compiler pipeline. Note that we had originally anticipated building a test harness and a set of unique test cases for each stage. However, this approach would have been prohibitively difficult given the narrow time-frame.

A comprehensive debugging output was added to the marshalling primitives in order to help locate errors. Information such as the expected and actual unmarshal type was printed, along with the details of the value kinds encountered during parsing to and from the byte-string.

4.5.1 Automated regression testing

Once program input and output was supported, the existing test system was extended to perform automated regression testing. Although this system was only implemented during January, it was at this stage that the more advanced features of Acute were being explored. The near-instantaneous feedback offered by this system was extremely valuable.

The testing was performed by a Perl script that iterated through each of the Acute source files in the test directory. Each Acute file had a header comment that was first parsed. The script then compiled and executed the source file. The comment contained the following optional fields:

`GRP` Specified the test case group membership. Individual groups could be tested.

`DOC` Specified a simple description of the test to be printed onto the console during execution.

`RET` Specified the expected `stdout` return value when the file was executed. Although this value was untyped (a simple string comparison was performed), this did not become a limitation during testing.

In the example in Figure 4.1, we define the successor function, and then print out the result of applying it to the value `42`. The `RET` value in the header specifies that the expected (textual) output value is `43`.

When all features were implemented, there were a total of fifty five tests, systematically covering all of the implemented language features. A common test idiom for the marshalling primitives was to use two sub-tests. The first test marshalled a value to the persistent store, whilst the second retrieved and unmarshalled it. The separate runtime instances ensured independence

```
(**
 * GRP  lambda
 * DOC  simple lambda
 * RET  43
 *)

let f = function x -> x + 1
in IO.print_int (f 42)
```

Figure 4.1: A test case for the successor function

```
let rec fib n =
  if n <= 2 then 1
  else (fib (n-1)) + (fib (n-2))
```

Figure 4.2: Naïve Fibonacci function

between the marshalling operations.

4.6 Performance

One of the main motivations for this project was to explore the performance benefits obtained by compiling the Acute intermediate language into O'Caml bytecode. In order to evaluate the performance of the bytecode, we benchmarked the execution time of the naïve Fibonacci function and Ackermann's function. The benchmark configurations are listed below. We use a constant value of `3` for the first parameter of Ackermann's function in order to obtain exponential time complexity. It must be noted that the benchmarks only provide a *very* general indication of the relative performances of the configurations.

acute-bc	Acute intermediate language, bytecode interpreted runtime
acute-ntv	Acute intermediate language, native runtime
abc	`abc` generated bytecode, modified O'Caml virtual machine
ocaml-bc	`ocamlc` generated bytecode, O'Caml virtual machine
ocaml-ntv	`ocamlopt` generated native code

The benchmarks were performed on an unloaded 1GHz dual Pentium III running PWF Linux. Each benchmark invoked the Unix `gettimeofday` function before and after running the test function, printing the epoch time value to `stdout`. A wrapper module was created to enable the Acute code

```
let rec ack m n =
  if m = 0 then n + 1
  else if n = 0 then ack (m-1) 1
  else ack (m-1) (ack m (n-1))
```

Figure 4.3: Ackermann's function

to access the system time information. Although the wrapper module increased the complexity of the testing (over using the Bash `time` command, for example), the benefit was that we could time the benchmarked function more precisely; our timing information did not include the runtime startup time.

A small set of shell scripts were created to automate the benchmarking process. Each benchmark source file was initially preprocessed in order to substitute in the correct value for the formal parameter n of the function. The file was then compiled before being executed three times with the resulting timing values recorded to a log file. A final script then invoked the Gnuplot program to create a graph of the benchmark results. Although this infrastructure took several hours to implement, the benefit was that the benchmarks could be *rapidly* and *repeatedly* run, with parameters adjusted as necessary. Removing the need for user intervention reduced the risk of any error occurring in the processing of the considerable amount of test data.

4.6.1 Analysis of results

The graphs are both of exponential time-complexity algorithms (as a function of the input argument value n). We therefore fit a regression function of the form $t = a^{n+b}$ to the timing data, where a and b are constants determined by the regression algorithm. The results are plotted on a logarithmic scale. We see that the gradients of the resulting functions are very similar, especially for the Fibonacci function where we have a greater number of data points. The horizontal displacements indicate that there is a constant factor difference between the execution times t. The ratios of the execution times for the functions are tabulated in Figure 4.6. We are unsure why the relative timings decrease for the Fibonacci function, although the results of such micro-benchmarks are clearly highly dependent upon the specifics of architecture implementations.

In both cases, we see that the native O'Caml code has the fastest execution time. The interpreted O'Caml bytecode is approximately an order of magnitude slower, with the `abc` generated bytecode again two to three times slower. The Acute bytecode runtime is between four and five orders

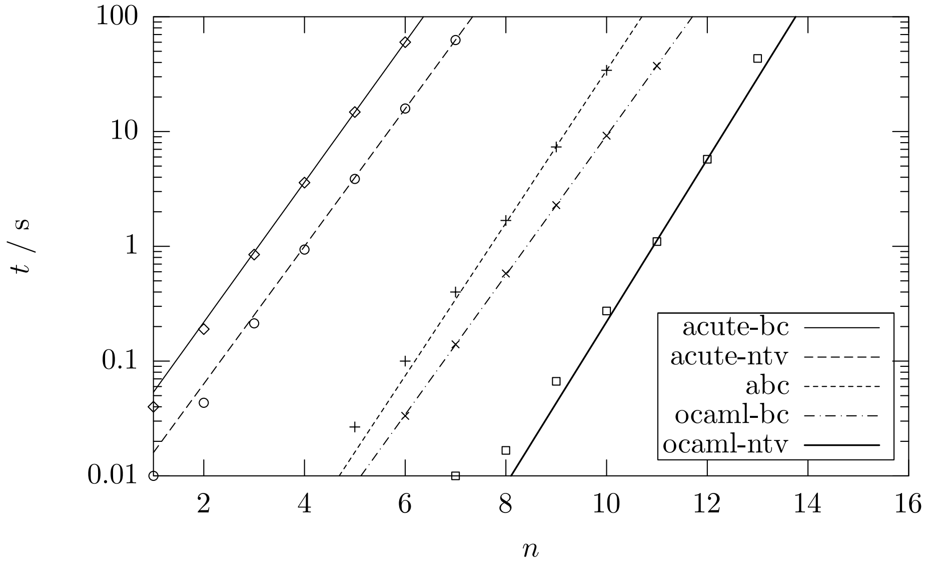


Figure 4.4: Ackermann's function benchmark results

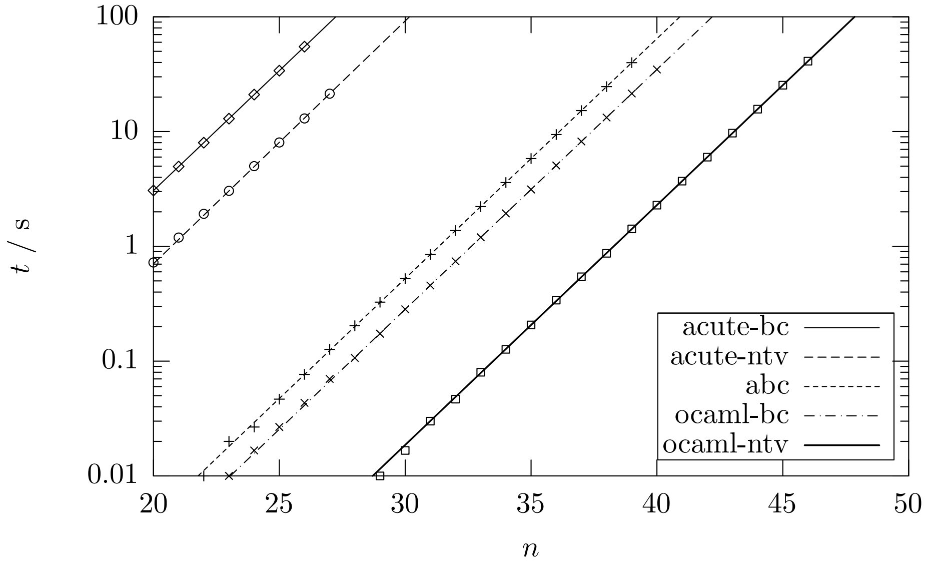


Figure 4.5: Fibonacci function benchmark results

Function	*O'Caml native*	*O'Caml bytecode*	*abc*	*Acute native*	*Acute bytecode*
Ackermann	1	41	110	27000	100000
Fibonacci	1	15	28	4800	21000

Figure 4.6: Relative execution speeds

of magnitude slower than the O'Caml native code. We see that the Acute native runtime adds approximately an order of magnitude speed increase.

The speed decrease between the `ocamlc` and `abc` bytecode can be attributed to the disabling of the 'threaded code' optimisation in the modified runtime, as well the lack of bytecode optimisations performed by `abc`. For example, `abc` performs no tail call elimination for tail recursive functions. Furthermore, each closure only accepts a single argument; this leads to a greater number of nested closure invocations for curried functions.

These benchmarks demonstrate that we have achieved comparable execution speed to `ocamlc` generated bytecode. Most importantly, we have increased the execution speed of Acute by *three to four orders of magnitude.* This is a considerable achievement.

We would have liked to explore the differences in runtime space efficiency between the different compilation methods. However, although the O'Caml virtual machine possesses a garbage collector interface, it would be relatively difficult to obtain meaningful results throughout the course of computation.

4.7 Exploring a larger example

Although the regression tests demonstrated that the Acute bytecode compiler was largely bug-free, we were keen to explore the feasibility of programming a larger, distributed example using the available features of Acute. The chosen example consisted of a graphical worm that eternally crawled between two windows, with the worm function being marshalled over TCP/IP whenever the screen figure reached the edge of its current window. A screen-shot of the one of the windows is shown in Figure 4.7. The white background has been shaded to aid viewing.

The worm code consisted of a single function that drew the image of the worm at the specified screen coordinates. The worm code was always marshalled with respect to a mark just above itself. This ensured that we cut the bindings to the graphics libraries, hence marshalling a minimal amount of code. We give the source code in Figure 4.8. The window code is contained in Appendix A.

Despite the non-trivial functionality, the complete example was programmed in fewer than one hundred lines of Acute code. Subjectively, this example was very easy to implement. However, it would have been useful to increase the number of primitive types supported by the language to include tuples and lists.

This example demonstrates that `abc` was capable of correctly compiling non-trivial code. Furthermore, the virtual machine extensions permitted the efficient runtime transfer of bytecode. The performance of the generated bytecode and runtime system was greatly improved over the original Acute runtime. In fact, a delay loop had to be added between each iteration to

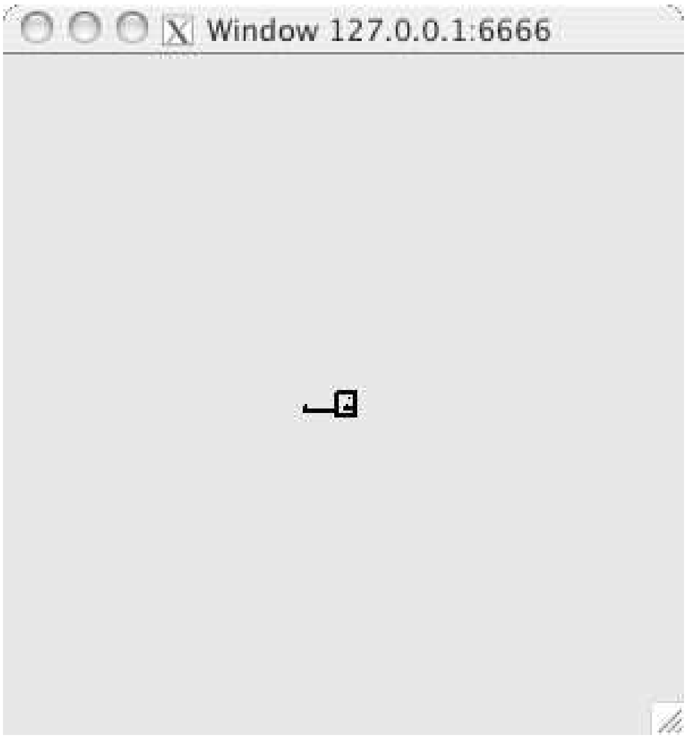


Figure 4.7: Screen-shot of the worm in action

prevent the humble worm from crawling too fast.

We see the true success of this project when we look back at the original starting point. We have brought powerful code marshalling facilities to the O'Caml virtual machine. Combined with high-level network access, these features begin to hint at the directions in which functional language development may progress to support distributed programming.

```
includesource "util.ac"

mark "MK"

(* worm function *)
let worm c x y =
  (* initialise *)
  Graphics.set_color c;
  Graphics.set_line_width 2;
  (* head *)
  Graphics.moveto x y;
  Graphics.lineto x (y-10);
  Graphics.lineto (x-8) (y-10);
  Graphics.lineto (x-8) y;
  Graphics.lineto x y;
  (* mouth *)
  Graphics.moveto x (y-7);
  Graphics.lineto (x-4) (y-7);
  Graphics.lineto (x-4) (y-6);
  (* eye *)
  Graphics.plot (x-3) (y-2);
  (* tail *)
  Graphics.moveto (x-8) (y-8);
  Graphics.lineto (x-22) (y-8);
  Graphics.lineto (x-22) (y-6);
  (* return next coordinate to plot worm *)
  Pair.pair (x+1) y in

(* bootstrap by marshalling worm and coordinates *
 * to one of the windows                         *)
Tcp.init 6668;
Tcp.send "127.0.0.1" 6666 (marshal "MK" worm);
Tcp.send "127.0.0.1" 6666 (marshal "MK" (Pair.pair 0 200))
```

Figure 4.8: Worm code

Chapter 5

Conclusions

We have exceeded our ambitious expectations ventured in the original project proposal. The standard compilation techniques facilitated the rapid design and implementation of a moderately efficient bytecode compiler. The initial grammar was quickly expanded to support a rather expressive fragment of the Acute language. The virtual machine extensions permitted the marshalling and rebinding of values, *including those of function types* – this is a significant improvement upon the standard O'Caml functionality. Furthermore, the emitted bytecode was shown to have a comparable performance to that generated by O'Caml. Throughout the implementation phase of the project, automated regression testing was used to great effect.

During the course of the project, the author gained a deep understanding of the O'Caml and Acute languages as well as the design and operation of the O'Caml virtual machine. Additionally, much was learnt about type theory, operational semantics and compiler design.

This project has demonstrated the feasibility of adding high-level marshalling facilities into production quality languages. Combined with rebinding, these primitives increase the control given to the designers of distributed systems. Furthermore, the clear semantic model facilitates reasoning about system properties.

Much research remains to be done in the area of marshalling and rebinding. From a language design point of view, the linear module structure of Acute is overly restrictive. In general, we could maintain sets of modules or use hierarchical module structures.

This project has taught us that manually scanning and patching bytecode is a rather complex operation. With regards to O'Caml bytecode, we could assist marshalling by retaining object file relocation information at runtime. More generally, we might compile to a more high-level bytecode format, maintaining explicit blocks of labelled closure code. For increased robustness, we could also include typing information. However, there are tradeoffs to be made with execution performance.

It is clear that distributed computation will remain a hard problem for some time. Servers will continue to crash and network connections will still fail. Yet perhaps we might begin to start understanding some of the systems that we build.

Bibliography

[1] The Java Language. See `http://java.sun.com`.

[2] IEEE Standard Glossary of Software Enigneering Terminology, 1990.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1999.

[5] Barry W. Boehm. A Spiral Model of Software Development and Enhancement, 1988.

[6] Didier Le Botlan and Alan Schmitt. The OCaml System – Implementation. Avaiable from `http://pauillac.inria.fr/~lebotlan/docaml_html/english/index.html`.

[7] Sylvain Conchon and Fabrice Le Fessant. *Jocaml: mobile agents for Objective-Caml*, May 1999.

[8] Xavier Leroy. Zinc. Available from `http://pauillac.inria.fr/~xleroy/publi/ZINC.ps.gz`.

[9] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed Closure Conversion, 1996.

[10] Dr Alan Mycroft. Lecture Notes on Compiler Construction for Part 1B of the Computer Science Tripos. Available from `http://www.cl.cam.ac.uk/Teaching/current/CompConstr`, 2004.

[11] Prof. Andrew M. Pitts. Lecture Notes on Types for Part II of the Computer Science Tripos. Available from `http://www.cl.cam.ac.uk/Teaching/current/Types`, 2004.

[12] Dr Winston W. Royce. Managing the Development of Large Software Systems, 1970.

[13] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. Design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp.

Appendix A

Example Code

In this section we present the Acute source code used for the window in the worm example.

```
includesource "util.ac"

mark "MK"

module Board :
  sig
    val init : unit -> unit
    val sleep : int -> unit
    val iter : (int -> int -> int -> ((int -> int -> int) -> int))
                 -> int -> int -> unit
  end =
  struct
    let init x =
      Graphics.open_graph " 300x300+250-250";
      Graphics.set_window_title "Window 127.0.0.1:6666";
      Graphics.auto_synchronize false

    (* fibonacci function to take some cycles... *)
    let sleep n =
      let rec fib x =
        if x = 1 then 1
        else if x = 2 then 1
        else (fib (x-1)) + (fib (x-2))
      in (fun x -> ()) (fib n)

    (* walk worm across board, marshalling when done *)
    let iter w x y =
      let rec it w x y _ =
        if x = Graphics.size_x () + 24 then
          (* marshal *)
          let addr = "127.0.0.1" in
          let port = 6667 in
         (Tcp.send addr port (marshal "MK" w);
          Tcp.send addr port (marshal "MK" (Pair.pair (0) y));
          Graphics.clear_graph();
          Graphics.synchronize ())
        else
          (* walk worm *)
          let _ = w (Graphics.white ()) (x-1) (y)
          in let p  = w (Graphics.black ()) x y
          in let _ = Graphics.synchronize ()
          in let x' = Pair.fst p
          in let y' = Pair.snd p
          in it w x' y' (sleep 20)
      in it w x y ()
  end

(* initialisation *)
let _ = Board.init ();
         Tcp.init 6666

in let rec go _ =
  (* block waiting to receive worm *)
  let worm = unmarshal (Tcp.recv ())
    as (int -> (int -> (int -> ((int -> (int -> int)) -> int)))) in
  (* ... and starting co-ordinates *)
  let p = unmarshal (Tcp.recv ())
    as ((int -> (int -> int)) -> int) in
  (* deconstruct co-ordinates *)
  let x = Pair.fst p in
  let y = Pair.snd p in
  (* abuse cbv semantics *)
  go (Board.iter worm x y)

in go ()
```

Appendix B

Typing Rules

This section gives the typing rules for the simply-typed lambda calculus intermediate language used within `abc`. These rules are heavily based upon those found in the Acute Technical Report [13].

The typing relation use two different kinds of environments: E maps module identifiers to signatures whilst Γ maps local identifiers to values. Where Γ is omitted we implicitly assume that this environment is empty.

B.1 Definitions

$$\frac{}{\mathrm{E}\ \vdash\ \mathrm{empty}\ :\ \mathbf{ok}}$$

$$\frac{\mathrm{E}\ \vdash\ \mathit{defs}\ :\ \mathbf{ok}}{\mathrm{E}\ \vdash\ \mathbf{mark}\ \mathrm{MK}\ \mathit{defs}\ :\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\mathrm{E}\ \vdash\ \mathbf{module}\ \mathrm{M}_M\ :\mathit{Sig}\ =\ \mathit{Str}\ :\ \mathrm{M}_M : \mathit{Sig}\\ \mathrm{E}, \mathrm{M}_M : \mathit{Sig}\ \vdash\ \mathit{defs}\ :\ \mathbf{ok}\end{array}}{\mathrm{E}\ \vdash\ \mathbf{module}\ \mathrm{M}_M\ :\mathit{Sig}\ =\ \mathit{Str}\ \mathit{defs}\ :\ \mathbf{ok}}$$

B.2 Modules

$$\frac{\mathrm{E}\ \vdash\ \mathit{Str}\ :\ \mathit{Sig}}{\mathrm{E}\ \vdash\ \mathbf{module}\ \mathrm{M}_M\ :\ \mathit{Sig} = \mathit{Str}\ :\ \mathrm{M}_M\ :\ \mathit{Sig}}$$

B.3 Signatures

$$\frac{}{\mathrm{E},\ \Gamma\ \vdash\ \mathrm{empty}\ :\ \mathrm{empty}}$$

$$\frac{\mathrm{E},\ \Gamma, x : T\ \vdash\ \mathit{str}\ :\ \mathit{sig} \qquad \mathrm{E},\ \Gamma\ \vdash\ v\ :\ T}{\mathrm{E},\ \Gamma\ \vdash\ \mathbf{let}\ \mathrm{x}_x\ =\ v\ \mathit{str}\ :\ \mathbf{val}\ \mathrm{x}_x\ :\ T\ \mathit{sig}}$$

$$\frac{\mathrm{E},\ \{\}\ \vdash\ \mathit{str}\ :\ \mathit{sig}}{\mathrm{E}\ \vdash\ \textbf{struct}\ \mathit{str}\ \textbf{end}\ :\ \textbf{sig}\ \mathit{sig}\ \textbf{end}}$$

B.4 Expressions

$$\frac{x \in \mathrm{dom}(\Gamma) \qquad \Gamma(x)\ =\ T}{\mathrm{E},\ \Gamma\ \vdash\ x\ :\ T}$$

$$\frac{C_0\ :\ T}{\mathrm{E},\ \Gamma\ \vdash\ C_0\ e\ :\ T}$$

$$\frac{op^n\ :\ T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T \qquad \mathrm{E},\ \Gamma\ \vdash\ e_j\ :\ T_j \qquad j \in 1 \ldots n}{\mathrm{E},\ \Gamma\ \vdash\ op^n\ e_1 \ldots e_n\ :\ T}$$

$$\frac{\mathrm{E},\ \Gamma, x : T\ \vdash\ e\ :\ T'}{\mathrm{E},\ \Gamma\ \vdash\ \textbf{fun}\ (x : T)\ \rightarrow\ e\ :\ T\ \rightarrow\ T'}$$

$$\frac{T_1 \approx T_2 \rightarrow T_3 \qquad \mathrm{E},\ \Gamma, f : T_1, x : T_2\ \vdash\ e\ :\ T_3}{\mathrm{E},\ \Gamma\ \vdash\ \textbf{funrec}\ (f : T_1),\ (x : T_2)\ \rightarrow\ e\ :\ T_2\ \rightarrow\ T_3}$$

$$\frac{\mathrm{E},\ \Gamma\ \vdash\ e_1\ :\ T\ \rightarrow\ T' \qquad \mathrm{E},\ \Gamma\ \vdash\ e_2\ :\ T}{\mathrm{E},\ \Gamma\ \vdash\ e_1\ e_2\ :\ T'}$$

$$\frac{\mathrm{E},\ \Gamma\ \vdash\ e_1\ :\ \mathrm{bool} \qquad \mathrm{E},\ \Gamma\ \vdash\ e_2\ :\ T \qquad \mathrm{E},\ \Gamma\ \vdash\ e_3\ :\ T}{\mathrm{E},\ \Gamma\ \vdash\ \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ :\ T}$$

$$\frac{\mathrm{E},\ \Gamma\ \vdash\ e_1\ :\ \mathrm{string} \qquad \mathrm{E},\ \Gamma\ \vdash\ e_2\ :\ T}{\mathrm{E},\ \Gamma\ \vdash\ \textbf{marshal}\ e_1\ e_2\ :\ \mathrm{string}}$$

$$\frac{\mathrm{E},\ \Gamma\ \vdash\ e\ :\ \mathrm{string}}{\mathrm{E},\ \Gamma\ \vdash\ \textbf{unmarshal}\ e\ \textbf{as}\ T\ :\ T}$$

$$\frac{\mathrm{M}_M \in \mathrm{dom}(E) \qquad \mathrm{E}(\mathrm{M}_M)\ =\ \textbf{sig}\ \mathit{sig}\ \textbf{val}\ \mathrm{x}_x : T\ \mathit{sig'}\ \textbf{end}}{\mathrm{E},\ \Gamma\ \vdash\ \mathrm{M}_M.\mathrm{x}\ :\ T}$$

Appendix C

Wire Format

This section specifies the full syntax of the wire format for marshalled values. Variable sized fields are prefixed by their byte length. This permits the encoding of arbitrary binary data.

Non-terminal symbols are italicised, whilst terminals are given in teletype font. Optional components are encased in curly parentheses {...}. We define the terminal alphabet Σ to be the set of byte values.

C.1 Integer literal

We define an integer literal *int* to be a signed, four-byte value.

$$\mathit{int} \in \Sigma^4$$

C.2 Identifier

We define an identifier *ident* to be a length prefix n, followed by a string of n binary characters.

$$\mathit{ident} \in \{s \in \Sigma^* \mid n \in \mathit{int} \ \wedge \ s = n\, s_1 \ldots s_n\}$$

C.3 Marshal value

A marshal value contains a type string, the mark with which the value was marshalled with respect to, a relocation table, zero or more module definitions and the value itself.

$$\mathit{marsh\text{-}val} ::= \mathit{type} \ \mathit{mark} \ \mathit{reloc} \ \{\mathit{module}\} \ \mathit{value}$$

C.4 Type string

The *type* corresponds to the pretty-printed string of the value type.

$$\mathit{type} ::= \mathit{ident}$$

C.5 Mark

The *mark* corresponds to the string value of the mark with which the value has been marshalled with respect to.

$$\mathit{mark} ::= \mathit{ident}$$

C.6 Relocation table

A relocation table *reloc* consists of a length prefix n, followed by the n relocation definitions *reloc-def*. Each of these entries is either a mark or a module definition. Semantically, the value *global-num* corresponds to the offset position of a module in the O'Caml global data block.

$$\mathit{reloc} \in \{s \in \Sigma^* \mid n \in \mathit{int} \;\wedge\; s = n\, \mathit{reloc\text{-}def}_1 \ldots \mathit{reloc\text{-}def}_n\}$$

$$\begin{array}{rcl} \mathit{reloc\text{-}def} & ::= & \mathit{reloc\text{-}mark} \mid \mathit{reloc\text{-}mod} \\ \mathit{reloc\text{-}mark} & ::= & \texttt{0}\ \mathit{ident} \\ \mathit{reloc\text{-}mod} & ::= & \texttt{1}\ \mathit{ident}\ \mathit{global\text{-}num} \\ \mathit{global\text{-}num} & ::= & \mathit{int} \end{array}$$

C.7 Module definition

A module definition *module* consists of a length prefix n, followed by n fields *mod-field*.

$$\mathit{module} \in \{s \in \Sigma^* \mid n \in \mathit{int} \;\wedge\; s = n\, \mathit{mod\text{-}field}_1 \ldots \mathit{mod\text{-}field}_n\}$$

$$\mathit{mod\text{-}field} \quad ::= \quad \mathit{value}$$

C.8 Value

A value may either be an immediate value *val-immed* or a closure value *val-clos*.

$$\mathit{value} ::= \mathit{val\text{-}immed} \mid \mathit{val\text{-}clos}$$

C.8.1 Immediate value

An immediate value *val-immed* is prefixed by the character `0` and then followed by an integer literal corresponding to the encoded value.

$$\textit{val-immed} ::= \texttt{0}\ \textit{int}$$

C.8.2 Closure value

A closure value *val-clos* is prefixed by the character `1` and then followed by the environment *env* and code *code*.

$$\textit{val-clos} ::= \texttt{1}\ \textit{env}\ \ \textit{code}$$

Environment

The environment *env* consists of a length prefix n, followed by n environment fields *env-val*.

$$\textit{env} \in \{s \in \Sigma^* \mid n \in \textit{int} \ \wedge \ s = n\, \textit{env-val}_1 \ldots \textit{env-val}_n\}$$

$$\textit{env-val} ::= \textit{value}$$

Code

The code part of the closure *code* consists of a length prefix n and a code offset m, followed by the binary code consisting of n four-byte values *code-val*. The code offset specifies which instruction the closure code pointer should reference within the block of code.

$$\textit{code} \in \{s \in \Sigma^* \mid n,\, m \in \textit{int} \wedge m \leq n \ \wedge \ s = n\, m\, \textit{code-val}_1 \ldots \textit{code-val}_n \ \}$$

$$\textit{code-val} ::= \textit{int}$$

Appendix D

Code Reachability

The following section defines a reachability relation for closure code. We use this definition during the marshalling of function values between runtimes.

For the analysis we assume that we are marshalling the high-level, labelled bytecode. However, the demonstrated technique readily applies to actual O'Caml bytecode instructions, although the relative references within the code cause the notation to become more complex. Additionally, we do not consider reachability over mutually recursive code blocks; such constructs are not present in `abc` generated bytecode.

D.1 Closure code

Let the code of a program at a given point in execution consist of a labelled sequence of instructions

$$\begin{array}{c} ins_1 \\ \vdots \\ ins_n \end{array}$$

A *closure block* c consists of a labelled sequence of instructions

$$\begin{array}{rl} c: & ins_p \\ & \vdots \\ & ins_q \end{array}$$

These instructions are delimited by the initial, labelled closure instruction and a final `return` instruction. More formally, these instructions satisfy the conditions

$$\begin{array}{c} p \geq 1 \\ q \leq n \end{array}$$

$$\mathit{ins}_q = \texttt{return 1}$$

D.2 Code reachability relation

Let C be the set of all such closures within the program code. Define the *code reachability relation* r_c over $C \times C$ as follows.

> The closure block pair $(c, c') \in C \times C$ is in r_c if and only if c contains an instruction of one of the forms

$$\begin{array}{r} \texttt{closure}\ \_,\ c' \\ \texttt{closurerec}\ \_,\ \_,\ c' \end{array}$$

We can visualise the relation r_c as a directed graph. We suspect that any such graph is acyclic for the supported fragment of Acute, although it is beyond the scope of this dissertation to attempt to prove this proposition.

D.3 Code reachability for runtime values

In order to marshal a value v, we must also marshal the *reachable code* that may be invoked from v. For this definition we require the transitive closure of r_c, which we denote by r'_c. If v is an immediate kind then no code is referenced. However, if v is a closure value, then code reachability proceeds through the immediately referenced code block as well as the field values.

$$\mathit{code\text{-}reach}\,(v) = \begin{cases} \{c\} \cup \{c' \in C \mid c\ r'_c\ c'\} \cup \bigcup_{v_i} \mathit{code\text{-}reach}\,(v_i) & \dagger \\ \{\} & \ddagger \end{cases}$$

† if v is a closure value with code block c and environment fields $v_1 \ldots v_n$
‡ otherwise

Note that we use a conservative, static approximation to the runtime code reachability. We assume that any closure instantiation instruction is executed. However, our approximation is certainly *safe*; it returns a superset of the code that will actually be invoked in any given execution run.

D.4 Implementation

We simply the marshalling implementation by transferring a single, contiguous block of instructions which contains all of the code blocks that are reachable from the value v.

$$\begin{array}{c} \mathit{ins}_j \\ \vdots \\ \mathit{ins}_k \end{array}$$

This block of instructions satisfies the properties

$$\begin{aligned} \mathit{ins}_j &= \min_i \mathit{ins}_i \in \bigcup \mathit{code\text{-}reach}\,(v) \\ \mathit{ins}_k &= \max_i \mathit{ins}_i \in \bigcup \mathit{code\text{-}reach}\,(v) \end{aligned}$$

Note that when we unmarshal code we allocate it on the C heap, and therefore it is unlikely to lie adjacent to those initial instructions loaded from an executable bytecode file. However, we suspect that such sets of closures are partitioned under r'_c and hence will never be simultaneously marshalled. This leads us to believe that the above implementation is valid.

Appendix E

Module Reachability

The following section defines a reachability relation for runtime module definitions. This relation is used when marshalling function values that reference modules.

E.1 Runtime module definition

For the analysis, we assume that modules are named. Consider a runtime module definition $m \in M$ to be a set of values

$$\{v_1, \ldots, v_n\}$$

E.2 Module reachability relation

Let M be the set of all modules definitions present within the runtime at a given point in execution. The set of module definitions that are immediately reachable from a value v is defined to be

$$\textit{immed-mod-reach}\,(v) = \{m' \in M \mid \texttt{getglobal}\ m' \in \bigcup \textit{code-reach}\,(v)\}$$

The *module reachability relation* r_m over $M \times M$ now proceeds through the fields of the modules as follows:

> The module pair $(m, m'') \in M \times M$ is in r_m if and only if
>
> $$m'' \in \{m' \in M \mid m' \in \bigcup_{v_i \in m} \textit{immed-mod-reach}\,(v_i)\}$$

We can visualise the relation r_m as a directed graph. This graph is acyclic for Acute modules due to the linear module definition structure.

E.3 Module reachability for runtime values

In order to marshal a value v, we must additionally marshal the set of *reachable modules*, consisting of all of those module definitions that may be accessed from v. For this definition we require the transitive closure of r_m, which we denote by r'_m. If v is an immediate kind then no modules are referenced. However, if v is a closure value, then module reachability proceeds through the immediately referenced code block as well as the environment field values.

$$
\mathit{mod\text{-}reach}\,(v) = \begin{cases} \mathit{immed\text{-}mod\text{-}reach}\,(v) & \dagger \\ \quad \cup \{m' \in M \mid m \in \mathit{immed\text{-}mod\text{-}reach}\,(v) \;\wedge\; m\ r'_m\ m'\} & \\ \quad \cup \bigcup_{v_i} \mathit{mod\text{-}reach}\,(v_i) & \\ \{\} & \ddagger \end{cases}
$$

† if v is a closure value with code block c and environment fields $v_1 \ldots v_n$
‡ otherwise

In the case of rebinding, we only marshal those reachable modules that occur after the marshal mark.

Appendix F

O'Caml Bytecode Instructions

This section gives a brief overview of the syntax and runtime behaviour of some of the more interesting bytecode instructions. Note that many of the instructions have specialised versions for small integer arguments. For example, there is a specific instruction for constructing a closure with one environment value.

An instruction consists of an initial 32-bit *opcode*, specifying the kind of the instruction (e.g. integer multiplication), followed by zero or more 32-bit arguments. Some instructions are of variable length. The actual opcode values are not given in this reference, although they may be found in the `opcode.ml` file in the standard O'Caml distribution.

All code offsets are signed values, and correspond to the distance in 32-bit words between the bytecode location of offset value and the referenced instruction.

F.1 Arithmetic operations

Arithmetic instructions typically consist of a single opcode. For example, an integer multiplication is performed by the `mulint` instruction. The runtime behaviour of this instruction is to destructively multiply the two argument values from the accumulator and the top of the stack, placing the result in the accumulator.

```
mulint
```

F.2 Block values

The instruction used to create a block value of kind k with n fields is given below. The kind is an integer value. The n field values are taken from the accumulator and the top of the stack at runtime.

```
makeblock n, k
```

F.3 Closures

F.3.1 Construction

The bytecode for constructing a non-recursive closure with n environment variables and function code at relative offset l is given below. At runtime, this instruction causes n values to be taken from the accumulator and the top of the stack in order to construct the closure environment. A pointer to the newly constructed closure block is then placed in the accumulator.

```
closure n, l
```

F.3.2 Application

The bytecode instruction used apply a closure to n arguments is given below. The runtime behaviour of this instruction is to destructively apply the closure reference contained in the accumulator to the n arguments residing at the top of the stack. The return value of the closure is placed into the accumulator. Note that this return value is also a closure if the original closure has only been partially applied.

```
apply n
```

F.3.3 Environment access

The code for accessing environment value i from within a closure is given below. The runtime behaviour of this instruction is to place environment value i into the accumulator.

```
envacc i
```

F.4 Recursive closures

F.4.1 Construction

A set of mutually recursive functions are represented by a single recursive closure. The multiple sections of bytecode are accessed via code pointers $l_1 \ldots l_m$. We also specify the environment size n.

closurerec m, n, l_1, $\ldots$ l_m

F.4.2 Application

A recursive closure is applied to arguments using the `apply` instruction. This causes the execution of the section of bytecode referenced from the first code pointer l_1. However, the closure code may place a reference to the containing closure into the accumulator by using the `offsetclosure` instruction. This permits recursive calls.

F.5 Global data

The `getglobal` instruction is used to access a field n of the O'Caml global data block. Note that we symbolically reference offsets within this block during compilation, delaying the calculation of actual positions until the linking stage. There is also an analogous `setglobal` instruction for entering the current accumulator block value into field n of the global data.

```
getglobal n
```

F.6 C Primitives

The `ccall` instruction is used to invoke a C primitive *prim* with n arguments. At runtime, the primitive is passed n instructions from the accumulator and the top of the stack. Note that the name *prim* is a symbolic; at link time this label is converted into an integer value. The named C primitive must be present at link time if static linking is being used.

```
ccall prim, n
```

Index

Project Proposal

In the following section we include a copy of the original project proposal.

John Billings
Queens' College
jnb26

Computer Science Tripos Part II Project Proposal

A Bytecode Compiler for Acute

October 21, 2004

Project Originator:	Dr Peter Sewell
Project Supervisor:	Dr Peter Sewell
Signature:	
Director of Studies:	Dr Robin Walker
Signature:	
Overseers:	Dr Jon Crowcroft Dr Ken Moody
Resources:	See Project Resource Form

Introduction

Many increasingly complicated systems are being fielded in the area of distributed computation. The design of such systems has created many novel challenges in the areas of development, deployment, execution and updates. Systems are deployed across multiple administrative domains, often with the requirement that multiple versions of programs can safely interact. Furthermore, the non-determinacy of distributed, parallel computation means that detailed analysis of system properties is often exceedingly complex. Current mainstream programming languages including ML, Java and C$^\sharp$ provide few facilities to address these problems.

Members of the Computer Laboratory Theory and Semantics Group have recently designed and implemented a high-level, distributed programming language called Acute[1], extending the O'Caml core. The language attempts to address many of the problems encountered with distributed computation. Facilities are provided for type-safe marshalling, as well as dynamic linking and rebinding. Acute can also create globally meaningful type names for abstract types, and enforce version constraints for rebinding. These features permit distributed infrastructures to be built as simple libraries.

Currently, a compiler for Acute has been written in Fresh O'Caml. This outputs compiled code in an intermediate language form, which is essentially the abstract syntax of Acute extended with closures. This intermediate language is then executed on an interpreter. However, this runtime is relatively inefficient, using pattern matching to perform reductions over the intermediate language.

The aim of this project is to implement a compiler for a small subset of the Acute intermediate language, targeting O'Caml bytecode. It is hoped that an appreciable performance gain will be seen when this bytecode is executed on the O'Caml virtual machine.

Technical Background

The Acute Language

"An Acute program consists roughly of a sequence of module definitions, interspersed with *marks*, followed by running processes... Marks essentially name the sequence of module definitions preceeding them. Marshal operations are each with respect to a mark; the modules below that mark are shipped and references to modules above that mark are rebound"[1].

As an illustration, consider the following example, adapted from [1]. A function is marshalled with respect to the mark `MK`. The reference to `M1.x` occurs before the mark, and hence will be rebound, whilst the reference to `M2.y` occurs after the mark and will therefore be marshalled. The marshalled value is transferred to a second runtime using the `IO.send` function.

```
module M1: sig val x:int end = struct let x = 0 end
mark "MK"
module M2: sig val y:int end = struct let y = 1 end
IO.send(marshal "MK" (fun () -> (M1.x, M2.y)))
```

The byte string is then unmarshalled in the second runtime. A runtime check is required to ensure that the unmarshalled value is of type `unit->int*int`.

```
module M1: sig val x:int end = struct let x = 0 end
module M2: sig val y:int end = struct let y = 1 end
(unmarshal(IO.receive()) as unit->int*int)()
```

Acute permits a high level of control for rebinding, supporting *version numbers* and *version constraints*. It is currently unclear whether the bytecode compiler will support these features.

The O'Caml Virtual Machine

The O'Caml virtual machine is based upon a stack architecture[2]. The stack contains the *values* for the bytecode operators. All values (for a given architecture) are the same size. A value is either an integer or a *block*. A block is a pointer to a data structure which also known as a block.

Blocks are dynamically allocated in memory, and consist of a header followed by zero or more fields. The header specifies:

- The size of the block
- Garbage collection information
- Constructor information

In O'Caml, functions are implemented as *closures*. The closure is allocated as block. The first field of the block points to the bytecode of the function, whilst the remainder of the fields correspond to the environment.

Marshalling Primitives

The project will require the implementation of `marshal` and `unmarshal` primitives. These will be implemented in C and compiled to a shared library. The standard O'Caml runtime system is able to dynamically load libraries and resolve references to user-implemented primitives prior to running the bytecode. The runtime also has a comprehensive API for the manipulation of runtime data[3].

In order to marshal a value, the bytecode representation must be encoded as a byte string. For an integer value, we can simply marshal the value on the stack. However, for a block value, we must marshal the referenced block. If this block is a closure, then we must also marshal the environment and the function bytecode.

At unmarshal time, blocks must be dynamically allocated. If the unmarshalled value is a closure then the references to the environment and the function bytecode will need to be modified to reflect their new positions in memory. There must also be a runtime type check. This suggests that the marshalled values will need to be tagged with their type and transferred along with a relocation table.

Should rebinding be implemented, then references from the marshalled values to modules will have to be altered to point to the memory locations of the existing modules.

Project Description

The first stage of this project will be to conduct a requirements analysis for the bytecode compiler. This will involve understanding the Acute language specification. Particular attention will be given to the reduction semantics, since these must be preserved in the bytecode translation. In addition, the O'Caml bytecode specification must be understood.

Several resources are available for the O'Caml bytecode specification, including online documentation and the O'Caml bytecode compiler source code. It is also possible to directly output bytecode mnemonics from the O'Caml bytecode compiler.

The next stage will be to design and implement a simple compiler for a very restricted form of the Acute intermediate language. There will be no need to write a parser for the intermediate language because the current Acute compiler implementation contains a module to perform this function. A light-weight testing framework will be concurrently developed. This resource should prove valuable when the more advanced features of Acute are implemented.

The compiler will be able to support the following language types:

- integers
- lambda abstractions
- lambda applications

The compiler can then be extended to produce bytecode that can reference external libraries. Once implemented, this facility will be used to link to the marshalling primitives. Additionally, it will hopefully permit the existing network modules to used, hence permitting marshalled values to be easily transferred between runtimes on different machines.

The next stage will be to implement the marshalling primitives. This will require a concrete marshalling format to be chosen. Additional changes may need to be made to the compiler in order for the runtime to retain sufficient information to permit the testing of type equality for marshalled values.

There is considerable scope for the extension of this project in the area of module reference rebinding for marshalled values. This will require the implementation of modules, although these can be considerably simplified by forbidding module initialisation. It will also be necessary to implement a facility to determine where a given module is located in relation to a specified mark.

Acceptance Criterion

The acceptance criterion will be the successful implementation of marshalling for the following types:

- integers
- lambda abstractions

For a demonstration of a successful implementation, it is hoped that the existing O'Caml network modules can be used to facilitate the transfer of marshalled values between runtime systems. However, a more primitive demonstration is possible, utilising the manual transfer of byte strings.

Starting Point

Much of this area of Computer Science is new to me. The Acute language specification is considerably more complex than anything covered in the Part 1B Semantics of Programming Languages course, and contains type theory that is not covered in the Part II Types course. Familiarisation with the new concepts will take some time. Furthermore, I have no knowledge of compilers beyond that presented in the Part 1B Compiler Construction course.

This project will make considerable use of Fresh O'Caml. I have no knowledge of ML beyond that which was taught in the Part 1A Foundations of Computer Science course. Therefore part of the initial stages of the project will involve studying the language, with particular emphasis on the module and type systems.

The marshalling primitives will be implemented in C. I have an intermediate knowledge of this language, having previously completed a sizeable project in C++. Therefore I anticipate little difficulty for this part of the implementation.

The parser module from the current Acute implementation will be used to obtain the abstract syntax tree from the intermediate language. All other code will be written by me.

Resources and Backup

Development will be conducted on the Computing Service Personal Workstation Facility under a GNU/Linux operating system. This system is fully supported by the Computing Service, hence reducing the risk of project disruption due to a system failure. Additionally, the required GNU compiler tool chain is already available on the PWF facilities. Therefore very little software configuration will be required before development can commence. The PWF facilities will primarily be accessed using the remote login facility from my own computer. However, any of the publicly accessible PWF machines may also be used.

The Acute source code is hosted on a Computer Laboratory server. During both the design and implementation stages I will frequently need to refer to this code. Additionally, I will need to make use of the Acute compiler modules that parse the intermediate language. The easiest way to view the authoritative version of the code will be by directly accessing the server. This requires a Computer Laboratory account. However, if this is infeasible, then it is possible to manually obtain a copy of the Acute source code.

Extra disc space may be required. Compiling Fresh O'Caml and Acute requires approximately half of the available disc space. Therefore, as a precaution against running out of disc space mid-way through the project, I would like to increase the available disc space from 250MB to 500MB.

A version control system will be used to store project files, permitting changes to be undone. Backups of the project files will be made to the Pelican archive server once per day. This process shall be automated. Additionally, weekly backups will be made to Compact Disc using my own machine's facilities.

Project Plan

I plan to have a functioning implementation of marshalling complete relatively soon (by mid-December), with the rest of the Christmas Vacation serving as ‘buffer-time’. This allows me to spend the first half of Lent Term implementing the proposed extensions to the project. The latter half of Lent Term can then be used to write up my dissertation.

Date	**Activity**
Fri 22 Oct 2004	**Proposal deadline**
Fri 05 Nov 2004	Understand Acute and O’Caml bytecode specifications
Fri 12 Nov 2004	Requirements analysis complete
	System design complete, including test harness
Fri 26 Nov 2004	**Compiler implementation complete for primitive language**
Fri 03 Dec 2004	*End of Michaelmas full term*
	Implementation of facilities to link to external libraries complete
Fri 17 Dec 2004	**Implementation of marshalling primitives complete**
Fri 14 Jan 2005	Start work on modules
Tue 18 Jan 2005	*Start of Lent full term*
Fri 21 Jan 2005	Start writing progress report
Fri 28 Jan 2005	**Module implementation complete**
	Progress report complete
Fri 04 Feb 2005	**Progress report deadline**
Fri 11 Feb 2005	Aim to have rebinding complete
Fri 18 Feb 2005	**Coding finished**
	Start writing dissertation
Fri 04 Mar 2005	Dissertation writing half-complete
Fri 18 Mar 2005	*End of Lent full term*
	First draught of dissertation complete
Fri 25 Mar 2005	**Submit draft dissertation to Dr Sewell**
Fri 08 Apr 2005	**Submit final dissertation to Dr Sewell**
Tue 26 Apr 2005	*Beginning of Easter full term*
	Submit dissertation
Fri 20 May 2005	**Dissertation deadline**
Tue 07 Jun 2005	*Written examinations start*
Thu 09 Jun 2005	*Written examinations end*
Fri 17 Jun 2005	*End of Easter full term*
Mon 20 Jun 2005	*Viva voca examinations*

Bibliography

[1] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, Viktor Vafeiadis
Acute: High-level programming language design for distributed computation, Design rationale and specification
http://www.cl.cam.ac.uk/users/pes20/acute/UCAM-CL-TR-605.pdf

[2] Didier Le Botlan, Alan Schmitt
The O'Caml System – Implementation
http://pauillac.inria.fr/~lebotlan/docaml_html/english

[3] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, Jérôme Vouillon
The Objective Caml System Release 3.08
Documentation and User's Manual
http://caml.inria.fr/ocaml/htmlman/index.html