Universität des Saarlandes
Fachrichtung 6.2 - Informatik
Lehrstuhl für Programmiersysteme

Diplomarbeit

Run-Time Byte Code Compilation, Optimization, and Interpretation for Alice

Christian Müller

März 2006

Angefertigt unter Leitung von
Prof. Dr. Gert Smolka

Betreut von
Dipl.-Inf. Guido Tack

Erklärung

Hiermit erkläre ich, Christian Müller, an Eides statt, dass ich die nachfolgende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, 29.03.2006

Abstract

Alice is a functional programming language that extends Standard ML with support for distributed and concurrent as well as constraint programming. One key feature of the system is that code is communicated in a high-level platform-independent format, called Alice Abstract Code, which is statically generated from Alice source code. The system ensures efficient execution by run-time compilation to native code. However, the native code compiler is not portable and it is hard to maintain.

This thesis develops a portable execution unit based on byte code. A new run-time compiler compiles the Abstract Code to Alice byte code that is executed by a register-based interpreter. Specialized instructions and dynamic code rewriting are used to speed up interpretation.

The module system of Alice imposes a strict separation of software blocks. This is only resolved at run-time as modules are linked dynamically. Thus, the run-time compiler has much more knowledge about the current state of the system than the static compiler. One optimization that is particularly effective at run-time is *inlining*. Two different forms of inlining (inlining of primitive operations, procedure integration) are investigated to reduce the overhead of procedure calls. The compilation strategy of the native code compiler is generalized to *selective compilation* in order to avoid compilation of rarely executed procedures.

The byte code system sets the new standard for platform-independent execution in Alice, starting from release 1.2. This thesis presents a careful analysis of the system and shows that its performance is competitive to the native code system.

Acknowledgments

I thank Professor Smolka for proposing this exciting project to me. I am impressed how much time and energy he devotes to discussions with students. I am particularly grateful to my adviser Guido Tack. He guided me through the project, always providing stimulating suggestions and ideas for further investigations. The implementation and the thesis itself profited a lot from his sense for simplicity and clarity.

Many thanks to Andreas Rossberg for sharing his expertise about Alice. He took always time to answer my questions. I also want to thank all other members of the Programming Systems Lab for making sure that working on my Fortgeschrittenenpraktikum and Diploma thesis was fun.

Finally, I am deeply grateful to my parents for all their patience and support.

Contents

1 Introduction

Alice [3] is a functional programming language based on Standard ML. It features support for distributed and concurrent programming as well as constraint programming. Alice programs are executed by a virtual machine whose open source implementation is based on a generic virtual machine library called SEAM [40]. The goal of this thesis project is to extend this system with a new execution unit based on run-time byte code compilation and interpretation. The new part of the virtual machine achieves efficient execution independent of the underlying hardware platform.

The Alice Programming Language

Alice offers rich support for *open programming*. As mentioned in "Alice Through the Looking Glass" [38] the main characteristics of open programming are modularity, dynamicity, distribution, and concurrency. These characteristics influence the architecture of the virtual machine.

Modularity means that software blocks are built separately and combined dynamically. A static compiler translates the modules into platform-independent code blocks that the virtual machine links at run-time. In general, software blocks can be imported and exported in a running program; this is meant by *dynamicity*. Thus, at run-time, the strict separation between modules is resolved. A dynamic compiler can capitalize on its knowledge about the imports of a module to generate efficient code.

The understanding of *distribution* in Alice is to communicate both data and code over networks. This requires a platform-independent representation of data and code that the virtual machine can handle. The implementation of the virtual machine has to support a wide range of platforms. In particular, the challenge is to offer equal performance for all platforms, such that programmers do not have to care on which architecture their Alice programs are actually executed.

The lightweight *concurrency* of Alice allows highly parallel execution of programs. The virtual machine coordinates concurrent execution of threads and supports implicit data flow synchronization based on the concept of *futures* [22].

1.1 Executing Alice Programs

The SEAM library offers a generic execution model in which customized execution units smoothly integrate. Figure 1.1 gives an overview about the execution model, which is described in the following.

SEAM's Execution Model

A *scheduler* coordinates concurrent execution of threads. Each thread maintains its own stack of activation records that are called *tasks* in SEAM. Each task references a concrete *task manager* that knows how to execute the task. A concrete *task creator* is

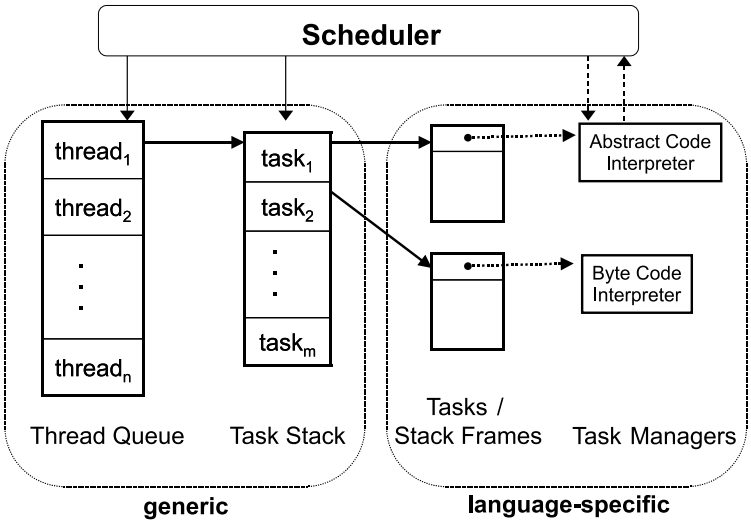


Figure 1.1: Seam's execution model

responsible for pushing new tasks onto the stack, which happens, for instance, when a procedure is called.

The framework explicitly supports mixed-mode execution. At run-time, the scheduler assigns the tasks to their dedicated task managers, using the reference that each task maintains. Two different approaches for executing Alice programs have existed before this thesis project.

(1) Abstract Code Interpreter

One key feature of Alice is that code is communicated in a high-level platform-independent format, called Alice Abstract Code, or abstract code for short. Thirty abstract code instructions suffice to express all kinds of Alice programs. The instructions are arranged as nodes of a graph – the abstract code graph – that explicitly encodes the control flow of a procedure. To sum up, the abstract code has a compact definition and, due to the high-level nature of the instructions, it is a memory-saving representation of Alice programs.

The static Alice compiler transforms each Alice program into a set of abstract code graphs. The virtual machine contains a task manager and a task creator for abstract code. The task manager is compact and serves as a practical reference for the semantics of the abstract code. However, interpreting abstract code is slow.

(2) Run-Time Compilation to Native Code

To speed up execution, there is a task manager for native code. To create a native code task, the system compiles abstract code graphs into native code functions at run-time. The execution unit just calls the generated native code functions.

Some typical benchmark procedures, for instance the Ackermann function, run up to 30 times faster when compiled to native code. However, the current run-time compiler has three severe drawbacks:

- It is not platform-independent, but only supports 32-bit x86 architectures.

- Debugging the native code system is extremely complicated.
- Generating efficient native code requires deep knowledge of the underlying hardware.

This thesis project aims at an alternative execution unit that is efficient, platform-independent, and easy to maintain. Thus the new approach falls in between the two existing ones.

1.2 New Approach: Byte Code System

In this thesis, we develop a new execution unit for the Alice virtual machine. The unit is based on byte code that is generated at run-time and executed on a byte code interpreter. Many optimizations are applied and the resulting system offers performance that is competitive to the native code system.

Alice Byte Code

Throughout the thesis, the term *byte code* stands for a sequential code representation, consisting of virtual machine instructions that can be executed on an interpreter. The Alice byte code is mainly a linearization and specialization of the abstract code. For each abstract code instruction, there is at least one corresponding byte code instruction. Additionally, there are specialized byte code instructions that offer efficient implementations of frequent special cases.

Execution Model

Byte code is executed by a register-based machine. All local variables of a procedure are kept in (virtual) registers that live in the current byte code task. As there is no restriction on the size of a task, there is no upper bound on the number of registers. All local variables can be mapped onto registers. The registers also store intermediate results of a computation.

The byte code instructions directly operate on virtual registers. In contrast to traditional stack-based interpretation, the instructions can randomly access the operands, and the register-based approach usually needs less instructions to implement a procedure. Whereas the debate on stack versus registers is decided for hardware [31], there is an ongoing discussion for software interpreters [17, 18] with a clear trend towards registers [41].

Task Execution: Interpreter

The byte code interpreter implements the machine model. In the standard execution model of Seam, the scheduler assigns a task to the interpreter that returns control directly after it finishes execution of the task. The execution might be interrupted by an exception or because of data flow synchronization during concurrent computations. Additionally, the scheduler periodically suspends the current thread to ensure fair distribution of processor time among all threads. Before the interpreter is left, the current machine state is saved in the task.

Task Creation: Compiler

Whenever a procedure is called, the scheduler delegates task creation to a concrete task manager. Depending on the configuration of the virtual machine, this might be

the byte code task manager. If the task creator finds out that the procedure has still an abstract code representation, it invokes the byte code compiler. The compiler traverses the abstract code graph in a depth-first manner and emits semantically equivalent byte code for each abstract code instruction. At the end, the compiler returns byte code, which implements the procedure, and the number of registers, which is needed during the execution. With this information, the task creator can generate a byte code task.

1.3 Byte Code Optimizations

To speed up execution, several optimizations are applied in the compiler and interpreter.

Register Allocation

As there is an unlimited number of virtual registers, the byte code compiler can use a one-to-one mapping from local variables to registers. This, however, leads to big tasks, and the bigger a task is, the more expensive is task creation. In addition, the tasks, or more precisely the task stacks, are subject to garbage collection, and huge task stacks slow down the collection. Register allocation is used to obtain a more clever mapping and since less registers are needed, the task size is reduced. A lightweight allocation algorithm, known as "Linear Scan Register Allocation" [34], is used.

Procedure Calls

We already mentioned that in the standard Seam model the interpreter executes a single byte code task and returns control to the scheduler. Performance benefits if the interpreter maintains control as long as possible. The byte code system uses dynamic (interpreter-driven) and static (compiler-driven) techniques.

The byte code interpreter dynamically tests if the procedure that is called has already a byte code representation. In this case, the interpreter maintains control, creates the new task and directly starts to execute it.

For some procedures, the byte code compiler can find out that they have a byte code representation. In this case, it generates specialized instructions that bypass the scheduler.

When the code representation can change during execution, for instance, from abstract code to byte code, a combination of static and dynamic techniques is used. The compiler tags the call instructions such that the interpreter can rewrite them if the dynamic test succeeds.

Procedure Integration

Optimized calls still produce more run-time overhead than executing straight-line code. The goal of *procedure integration* – better known as *inlining* – is to eliminate calls altogether. The byte code compiler automatically integrates procedures that do not exceed a fixed size limit into their caller. As the integration analysis happens at run-time, one main challenge is to balance power and overhead of the analysis. Another challenge is to efficiently simulate parameter passing from the caller to the embedded callee and back.

Selective Compilation

The compilation strategy of the native code system is to compile each procedure before its first execution. As the strategy is implemented as lazy evaluation (compilation) of code, there is no way to postpone compilation further. The byte code system uses a more flexible technique, which we call *selective compilation*. This allows to formulate compilation as transition between different code stages. Transitions are controlled by arbitrary boolean conditions. This way, it is easy to express the condition that procedure p is not compiled until the n-th execution. Of course, the transition system can contain more than two stages. One can think of a system with k compilation steps, and in each step more and more and more optimizations are applied.

1.4 Contributions

Before this thesis project, there was a huge performance gap between platform-independent interpretation and platform-dependent run-time compilation. The byte code system developed in this thesis project is platform-independent and its performance comes close to the native code system. The implementation is fully operable and it is part of all Alice releases starting from release 1.2.

We investigate procedure integration, selective compilation, dynamic tests, and code rewriting in the byte code system. All existing execution units are evaluated carefully, with special focus on the optimizations of the byte code system. In general, the byte code system is a good platform to experiment with new optimizations. As it is implemented in the high-level language C++, it is much easier to modify and extend than the native code system.

There is few documentation about SEAM and the Alice language layer. Brunklaus and Kornstaed give an overview [13]. Chapter 2 and 3 contain detailed information about the virtual machine, which helps to close the gap between the conceptual view and the actual implementation.

1.5 Structure of the Thesis

Chapter 2 and 3 present the architecture of the Alice virtual machine and explain properties of the abstract code and native code. In Chapter 4, we introduce the machine model and instruction set of the byte code interpreter. Chapter 5 develops the translation phase from abstract code to byte code. Chapter 6 focuses on optimizations that are needed to achieve good run-time performance. Implementation-specific aspects are discussed in Chapter 7. Chapter 8 compares the abstract code interpreter, the byte code and the native code system and evaluates interesting system properties like compile time and code size of Alice applications. A conclusion of the thesis is given in Chapter 9.

2 The Alice Virtual Machine

Alice programs are executed by a virtual machine that is built on top of the virtual machine library Seam (Simple Extensible Abstract machine). The library provides generic services that ease implementation of virtual machines for modern programming languages. It offers an abstract data model as well as an abstract execution model that explicitly supports mixed mode execution. The architecture of Seam is originally described by Brunklaus and Kornsteadt as "A Virtual Machine for Multi-Language Execution" [13]. This chapter gives an overview about the framework and details the Alice specific aspects that are embedded into the model as a so called language layer.

2.1 Seam

Seam provides generic services that are common to a wide range of virtual machine implementations. The two central generic components are the *scheduler* and the *abstract store*. Language-specific aspects are parameterized and language implementors have to equip their implementations with defined interfaces. All units that are specific to a language L are called *language layer* for L.

2.1.1 Scheduler

The lightweight threads, offered by Alice, directly build on Seam's concurrency model. The scheduler coordinates concurrent execution of several threads. Every thread maintains its own stack of activation records. An activation record is called *task* in Seam's terminology.

As depicted in Figure 1.1 (page 2), the scheduler coordinates execution on two levels. The first level is fair distribution of processor time between threads. All threads reside in a queue. Whenever a fixed time slice has elapsed, a flag is set to a global *status register* that issues the scheduler to suspend the current thread and choose the next thread in a round-robin fashion. Thread coordination is generic and language implementors only have to ensure that the status register is checked periodically.

The second level is coordination of tasks inside a thread. The scheduler delegates creation and execution of tasks to language-specific execution units. A global register bank serves for language independent communication between tasks. Seam defines a generic interface for execution units, but does not prescribe a specific execution model since this is, in general, highly language-specific.

Execution Unit

A language-specific execution unit that respects Seam's generic interface consists of two components: a *task creator* and a *task manager*, also called *interpreter*. The task creator sets up a new task from a first class computation that consists of a piece of code and an associated environment. For instance, a procedure call is prepared by issuing a

task creator to push a new task. Each task contains a reference to its interpreter. The task abstraction allows for simple language inter-operation and – more importantly in the context of this project – for multiple execution modes. At run-time, several different execution units coexist. As each task knows its associated task manager, the scheduler can easily assign a task to the right task manager.

To implement an additional execution unit for a specific kind of code, one has to write a task creator and an interpreter that both respect the scheduler interface.

Tasks can not only represent computations that the user of the system triggers. They are also used internally by the virtual machine to implement services. There is for instance a service to minimize (parts of) the data graph in the abstract store (Tack [43]). This service defines its own task creator and manager and can therefore be controlled by the scheduler.

Exceptions

Exceptions permit a structured form of jump and a save way to transmit data to the jump target. In general, the exception mechanism builds on two constructs: an exception can be *raised* and *handled*. Raising an exception means to abort the current computation and to pass the exception value to the run-time system. The run-time system searches for a dynamically enclosing block that knows how to handle the exception. The task stack constitutes the dynamic scope and is therefore searched for an appropriate handler.

SEAM's scheduler features generic support for exceptions. When a concrete execution unit raises an exception, it writes the exception value to the scheduler register R_{exn}. An exception handler is a pair: its first field contains a reference to the task that pushed the handler; the second field contains the *handler procedure* that specifies the formal argument and the code of the handler. Every thread maintains an extra stack for all handlers. When an exception occurs, the scheduler looks for an appropriate handler in the handler stack and skips all tasks that are above the task referenced by the handler (called *current task* in the following). This is especially efficient when the exception aborts a recursive procedure that has allocated many tasks.

SEAM does not prescribes the exact representation of the handler procedure. Instead, every execution unit implements a special task creator that the scheduler invokes to start execution of the handler procedure. The task creator binds the formal argument of the handler to the content of R_{exn} and modifies the current task, such that the handler procedure is executed when the scheduler assigns the task to its interpreter.

2.1.2 Abstract Store

SEAM permits an abstract view on memory that allows high-level modeling of language-specific data structures. The abstract store holds a *data graph* that has a designated entry point, called *root set*. The graph is constructed from two basic node types: *chunks* are raw byte data of arbitrary length, and *blocks* contain a number of directed edges to other nodes. In order to make use of efficient native operations, *integers* are provided as an additional node type. The library already defines some high-level data structures, for instance, (weak) maps, dynamic arrays, stacks and queues. Each of these data structures is defined on the abstract store and can fully use all its services. Two major services are garbage collection and *pickling*. Additionally the abstract store

provides *transients* that can elegantly express synchronization and communication in concurrent systems.

Garbage Collection

Garbage collection frees memory that is occupied by unreachable store nodes. SEAM uses generational garbage collection [46]. The starting point for the collection is the root set. Language implementors only have to know about the root set to endow their language with a high-quality garbage collection. The implementation itself also benefits from this service. SEAM is implemented in C++, which does not offer automatic memory management. Implementation of language-specific services often requires many temporary data structures. Using SEAM's predefined data structures, which are subject to the internal memory management, facilitates implementation and maintenance.

Pickling

The data graph in the abstract store can be transformed into an external byte string, called *pickle*. The pickle is stored in a file or transmitted over the network, allowing for persistent storage of data graphs as well as platform-independent data exchange. The *unpickler* reconstructs an isomorphic data graph from the pickle. Tack, Kornstead and Smolka present "Generic Pickling and Minimization" [44] as it implemented in Alice.

The abstract store is a generic memory model. However, nodes in the data graph can contain platform-dependent data. If the language implementor wants to use the pickling service, he has to provide means to transform the platform-dependent representation of nodes into a platform-independent one. So for every *concrete representation* there must be a handler that translates it to an *abstract representation*. For instance, the concrete code representation might be native machine code for Intel processors. This code is of course invalid for the PowerPC. Before code can be pickled, it has to be transformed into an abstract, i.e. platform-independent, representation. For Alice, this is the *Alice Abstract Code*, which is presented in Section 3.1.

Transients

The concurrency model of SEAM realizes synchronization and communication between threads with *transients*. Transients are placeholders for yet undetermined values. When a computation hits a transient instead of an actual value, it returns control to the scheduler that blocks the thread until the value is determined. There are several kinds of transients of which *futures* and *by-need* values are described in this section.

The notion of futures first occurs in Multilisp [22]. In Alice, a future is a transient that might become a value in the future. It maintains a queue of blocked threads that are rescheduled when the future is replaced by an actual value. A by-need is a transient that is associated with a first class computation. So it contains the code and all data required to compute the actual value. When a thread requests the actual value, the by-need is transformed into a future on which the requesting thread blocks. At the same time, the run-time system creates a new thread with two tasks on its task stack. The first task spawns the first class computation to compute the actual value. The second task explicitly replaces the by-need with the result.

Niehren, Schwinghammer and Smolka give a more formal treatment of futures for statically typed languages. They investigate a concurrent λ-calculus with futures [32].

2.2 The Alice Language Layer

The language layer defines all language-specific aspects of the Alice virtual machine. It models all Alice values on SEAM's abstract store, and it contains three different execution units for *Alice Abstract Code*, native code and byte code. Code also lives inside the abstract store, which has some interesting implications for the actual representation.

2.2.1 Alice Values

All Alice values, which are directly modeled on the abstract store, are listed in Table 2.1. For each value type there is an example given that relates the internal values to the external representation in the source language.

value	**example**
integers	`1234`
tuples	`(x,2)`
closures	`fn x => x+y`
transients	`lazy (x+1)`
strings	`"Hello, world!"`
real numbers	`123.456`
reference cells	`ref 23`
arrays and vectors	`#[1,2,3,4]`
records	`{a=3, b="string"}`
tagged values	`MyConstructor (x,y)`
constructed values	`MyException "error"`

Table 2.1: Alice values

The SEAM library defines integers, tuples, strings, reals, closures, and several types of transients, for instance, futures and by-needs. All other values are defined in the Alice language layer. Due to space restrictions, this thesis will not describe how these values are modeled on the abstract store. Details on the data layout can be found in Guido Tack's Diploma thesis [43]. We concentrates on a restricted value set, consisting of integers, tuples, closures, tagged values and exceptions. The *restricted value set* is then used for a concise description of Alice Code.

Integers. Integers are represented as a special node type in the abstract store, so they are directly supported by SEAM. They are represented in a way that allows the use of native arithmetic operations. Alice also provides big integers values, which can become arbitrarily large and cannot be mapped on hardware. In the following, we always mean the small integer values when we just speak of integers.

Tuples. Tuples directly correspond to the mathematical concept. Internally, they are represented as a block. So tuples allow a more abstract view on blocks.

Closures. A closure is a pair consisting of self-contained piece of code and a global environment that contains all free variables occurring in the code. In the example given in Table 2.1, the variable `y` is free and will therefore be bound by the global environment.

Tagged Values. The type system of Alice supports `datatype` declarations.

```
datatype tt = A of int | B
```

The static Alice compiler assigns an integer tag to each constructor. A constructor values of the form `(A 34)` is a block that contains the tag for `A` and all data fields. For efficiency, nullary constructors are represented as integers since the tag is the only information they store.

Exceptions. Exceptions are defined as constructed values. These values implement Alice `exttype` declarations, which are data types that can be extended dynamically. Instead of an integer tag, they contain an arbitrary block that uniquely identifies them.

2.2.2 Alice Code

The Alice language layer defines three types of code. This section lists the code forms and discusses some principle properties. Details on the representation and the instructions are given in Chapter 3 and Section 4.3.

Alice Abstract Code

One key feature of Alice is that code can be communicated in a platform independent format, called *Alice Abstract Code*, or *abstract code* for short. The static Alice compiler compiles the source language into abstract code. This code form is the external storage format for Alice Code, used in pickles. So it is the only code form that survives the run-time of a program.

The abstract code is defined by an Alice data type declaration and internally represented as tagged values. Conceptually, the code makes up a graph whose nodes encode the instructions of the abstract code. The instructions are high-level, which means that one instruction comprises many execution steps. This yields a compact definition of the instruction set. Additionally, the code consumes less memory, which allows fast code exchange over a network connection.

Native Code

To achieve fast execution speed, the run-time system compiles the abstract code to native code, before a procedure is executed for the first time. The native code compiler does only work on 32-bit x86 processor architectures.

Byte Code

Run-time compilation to byte code is the platform-independent way to speed up execution. Alice byte code can be roughly described as a linearizion and specialization of the abstract code. The definition of a byte code for Alice and the development of an efficient interpreter and an optimizing run-time compiler from abstract code to byte code are the main contributions of this thesis.

Both byte code and native code are represented in a linear fashion and occupy a contiguous range in memory. This memory region has a meaning only for an interpreter or a hardware processor, respectively. We call this kind of representation *binary code*.

Code as Data

All code forms are modeled on SEAM's abstract store. Since code is just another form of data, it is subject to garbage collection and can be pickled. The resulting challenges are different for abstract code and binary code.

Abstract code consists of values that the Alice language layer provides. So neither garbage collection nor pickling causes any trouble because they can simply be used as for any other Alice values.

The binary code formats native code and byte code occupy a contiguous range in memory. This can either be modeled as a block or a chunk. Code typically contains only a small number of references to other store nodes. It is a reasonable design decision to choose chunks for the representation of binary code. The only difficulty with chunks is that the garbage collector regards them as (atomic) leaf nodes and does not update references from binary code to other store nodes. Therefore, binary code is represented as a pair of a code chunk and an *immediate environment* that is a block that contains references to store nodes. Native code and byte code access all immediate values indirectly via indexes into the immediate environment.

As code chunks live in the abstract store, they are subject to garbage collection. For this reason, binary code has to be *position independent*, such that all addresses are still valid after the garbage collector moved the code to a different location.

Native code is by its nature platform-dependent. The abstract code is the abstract representation of native code. In order to make use of pickling, there must be a handler that transforms native code into abstract code. In general, this backward transformation is quite complicated, especially when advanced compiler optimizations like procedure integration are involved. The current architecture keeps both code representations in memory. The execution unit uses the native code whereas the pickler takes the abstract code. This is the simplest solution although it wastes some memory. Byte code is represented in the same way.

Putting all pieces together, the concrete representation of binary code is a triple of a code chunk, an immediate environment and an abstract code graph (as abstract representation).

2.2.3 Execution Units

There is an interpreter that can directly execute the abstract code. It is compact and serves as an executable reference for the semantics of the abstract code. Since the abstract code is not designed for efficient execution and the interpreter does not apply any optimizations, interpreting abstract code is very slow. To speed up execution, a run-time compiler transforms every procedure into native code or byte code when it is invoked for the first time. Byte code is executed on a special byte code interpreter (Chapter 4). The native code compiler creates self-contained native functions that are called by a native code interpreter. The scheduler coordinates the interpreters and assigns the tasks to them. Many optimizations are applied to return control to the scheduler as seldom as possible.

Executing native code is on average three to five times faster than interpreting the abstract code, and there are some extreme cases like Ackermann's function that is more than thirty times faster when using native code. The performance of the byte

code system can compete with the native code system. Detailed benchmarks are given in Chapter 8.

Exceptions

The exception mechanism of Alice builds on SEAM's generic exception service. To understand how the exception mechanism works, let us consider the following example:

```
exception ZERO

fun mlist' (nil,a)    = a
  | mlist' (0::_,_)   = raise ZERO
  | mlist' (x::xr,a)  = mlist' (xr,x*a)

fun mlist xs = mlist' (xs,1) handle ZERO => 0
```

Procedure `mlist'` multiplies all elements of a list. If the list contains zero, it aborts computation and raises exception `ZERO`. Procedure `mlist` contains code to handle this exception. The static Alice compiler creates a handler procedure that closely resembles to the following pair:

```
(x, case x of ZERO => 0 | _ => raise x)
```

The first component specifies the formal argument `x` that binds the exception value. The handler code first checks if it can handle the current exception. For our example, the handler is only responsible for the exception `ZERO`. In all other cases, the exception is re-raised, which means that it is passed to the next handler on the handler stack.

Calling Convention Conversion

A calling convention defines how parameters are exchanged between procedures. Following the standard terminology, the procedure that invokes another procedure is the *caller* and the procedure that is called is the *callee*.

SEAM offers a global register bank for generic parameter passing, but it does not prescribe a special calling convention. Conceptually, there are only unary procedures in Standard ML and this also extends to Alice. If a caller wants to pass several parameters, it packs them into a tuple and the callee selects all elements that it needs. In the majority of cases, the tuple construction is superfluous because the callee is interested in all components and does not use the tuple. The Alice virtual machine avoids tuple construction whenever possible by using a dynamic calling convention conversion. The general idea is that the receiver is responsible to convert the arguments into the format it expects. There are four cases:

- *match:* The caller supplies exactly the number n of arguments that the callee expects. In this case, the callee reads registers 1 to n from the global register bank without conversion.
- *mismatch – tuple expected:* The caller offers $n > 1$ arguments, but the callee expects exactly one argument. In this case, the callee constructs a tuple consisting of all arguments in the register bank.
- *mismatch – discrete arguments expected:* The caller supplies one argument, but the callee expects $n > 1$ arguments. In this case, the callee has to deconstruct the tuple that is in register 1.

- *mismatch – no argument supplied:* The caller does not supply an argument, but the callee expects one. In Alice, callers do not supply tuples of arity zero. The empty tuple is uniquely represented as integer number zero. So the callee can construct the value when it needs it.

The interpreter implementor has to ensure that every procedure checks the number of arguments in its prelude. Such a check is also needed whenever a callee returns to its caller and uses the register bank to pass some arguments. An optimizing compiler can generate code that skips the tests if the compiler detects that there is always a match.

3 Alice Code

This chapter describes the code formats that have already been present before this thesis project. It presents the *Alice Abstract Code* and gives an overview about the native code system. The conclusion of the chapter motivates why a byte code system for Alice has some significant advantages over a native code system.

3.1 Alice Abstract Code

The first part of this section addresses the format of the abstract code, whereas the second part deals with the instructions. At the end, there is a short discussion of advantages and disadvantages of this code representation.

3.1.1 Format

Alice represents code as a directed acyclic graph. The acyclic structure is achieved by transforming all loops into recursion. The nodes of the graph encode the instructions of the abstract code. Except for the root node, each node has either one or two predecessors. A node has $n \geq 0$ successors. The code graph explicitly encodes the control flow, which means that each instruction knows about its continuations.

The static Alice compiler removes all information about types. Static type safety guaranties that the abstract code instructions operate on correctly typed arguments. In particular, there are no polymorphic instructions, which means that the instructions do not have to test the type of its arguments.

Alice Procedures

Code only occurs as procedure bodies. Each *procedure code* contains the root of its abstract code graph, a list of formal arguments and some additional information to ease compilation (see Chapter 5 and 6). Procedure code is parametrized over global values,

$$
\begin{array}{lll}
\textit{procedure\_code} & ::= \{ & \\
 & \texttt{root:} & \textit{instr} \\
 & \texttt{formalArgs:} & \textit{def}^* \\
 & \texttt{numberOfIds:} & \textit{int} \\
 & \texttt{outArity:} & \textit{int option} \\
 & \texttt{subst:} & \textit{id} \rightarrow \textit{value option} \\
 & \texttt{liveness:} & (\textit{id} \times \textit{int} \times \textit{int})^* \\
 & \} &
\end{array}
$$

Figure 3.1: procedure code with annotations

which means that these values are indirectly accessed via free variables. An *Alice procedure* is a closure that contains a procedure code and an environment of values for the free variables. Notice that procedures are the only abstraction mechanism in the abstract code. The static Alice compiler transforms every higher-level concept, like functors or structures, into procedures.

Static Single Assignment Form

The abstract code is in static single assignment (SSA) form. SSA is a property of intermediate languages that makes compiler optimizations clean and efficient. If an intermediate language is in SSA form, then every variable has a single definition site. Single-assignment is a static property because in a general control flow graph the assignment might be in a loop. SSA form was developed by Wegman, Zadeck, Alpern, and Rosen [4, 37] for efficient computation of data flow problems, such as global value numbering and detecting equality of variables.

Given an arbitrary program, a variable with name u might be used for several unrelated purposes. Transformation to SSA form removes all superfluous dependencies and thus makes optimizations more powerful. Let us consider the following program:

$$\begin{aligned} u &\leftarrow a + b \\ v &\leftarrow u + 1 \\ u &\leftarrow a + 1 \end{aligned}$$

It is not directly clear from the names that the definition of v does not depend on the second definition of u. If we choose new names such that each assignment gets a fresh name, the dependencies become obvious.

$$\begin{aligned} u_1 &\leftarrow a + b \\ v_1 &\leftarrow u_1 + 1 \\ u_2 &\leftarrow a + 1 \end{aligned}$$

For straight-line code, the transformation is quite easy because it only means to α-rename the program. The situation becomes more difficult if two control flow edges join, as depicted in Figure 3.2.

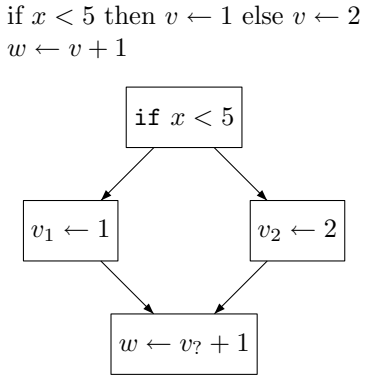


Figure 3.2: control flow graph for an `if` expression

Following the simple renaming strategy, it is not clear whether $v_? = v_1$ or $v_? = v_2$. In fact, this cannot be decided statically. SSA form solves the problem with a notational trick, called ϕ-function. These pseudo-assignments are introduced at the beginning of

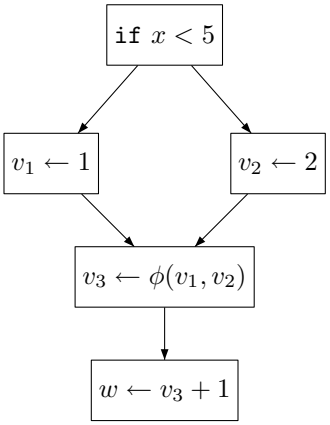


Figure 3.3: control flow graph with ϕ-function

a basic block (compare Figure 3.3). If the left control flow branch is taken, $\phi(v_1, v_2)$ has the value v_1, and if the right control flow branch is taken, it has the value v_2. Intuitively, ϕ-functions synchronize variable names at control flow join points such that the SSA property (single definition point) is maintained. The simplest approach for ϕ-placement, is to place ϕ-functions for *all* variables at *every* join point in the control flow graph. This approach introduces many unnecessary ϕ-functions. Bilardi and Pingali describe a range of algorithms for optimal ϕ-placement and compare their efficiency [11].

SSA form is typically used as intermediate representation for imperative languages. The functional programming community prefers the λ-calculus and *continuations* as intermediate language. Andrew Appel pointed out the close relationship between the two representations in his article "SSA is Functional Programming" [6].

Executable SSA Form

If we want to interpret code in static single assignment form, ϕ-functions cause some trouble. Whenever the execution takes a branch, it has to remember its number in order to be able to evaluate ϕ-functions. Interpreting static single assignment form was recently proposed by Ronne, Wang and Franz [45]. In contrast, Alice uses a modified version of SSA form without ϕ-functions. The single assignment property, which is that every variable has a single definition site, is weakened to

> *on every path each variable is defined at most once*

This definition only works because all paths in the abstract code graph are acyclic. The static Alice compiler eliminates ϕ-functions by trying to assign the same variable names on each branch. If this fails for a certain branch, an explicit move instruction is inserted at the end. An example is given in Figure 3.4. The additional move instruction is depicted in a dashed node. For simplicity, we assume that the compiler cannot directly assign v_2 to 1 on the left branch.

As all ϕ-functions are converted from a pseudo instruction into directly executable code, the abstract code can run on an interpreter without modification.

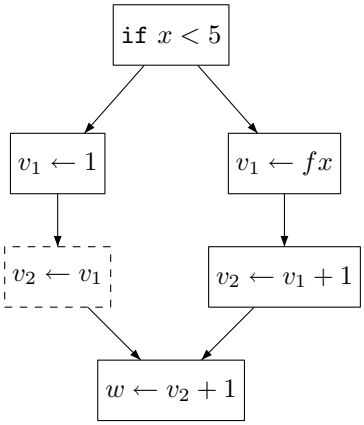


Figure 3.4: control flow graph with explicit assignment

3.1.2 Instructions

The set of abstract code instructions decomposes into five groups:

(1) allocation of data structures

(2) access to data structures

(3) conditionals

(4) procedure application

(5) exceptions

The remainder of this section defines the abstract code instructions for the restricted value set, which composes of integers, tuples, closures, tagged values, and exceptions. A record-like notation is used for each instruction. This notation aims for a comprehensible presentation. The actual implementation represents the instructions as data type values. They can be obtained by stripping the record annotations from the instructions, preserving the order of the constituents.

Identifiers

The abstract code explicitly differentiates between defining and applied occurrences of identifiers. Let *id* be the set of identifiers and *value* denote the set of all Alice values. Defining occurrences of identifiers can have the following form:

$$\begin{array}{lcll} def & ::= & \texttt{IdDef} & \{\texttt{id} : id\} \\ & | & \texttt{Wildcard} & \end{array}$$

There is either a local definition of an identifier or a wildcard. A wildcard is only a hint that the result of the definition is not used.

There are three forms of applied occurrences:

$$\begin{array}{lcll} ref & ::= & \texttt{Global} & \{\texttt{id} : id\} \\ & | & \texttt{Immediate} & \{\texttt{value} : value\} \\ & | & \texttt{Local} & \{\texttt{id} : id\} \end{array}$$

Global identifiers refer to locations in the closure. Local identifiers refer to locations in the current task, i.e. stack frame. The static compiler propagates each constant value

to its use site. The abstract code refers to them as immediate values. Each abstract code instruction is parameterized over *ref* and *def*, respectively. This yields a compact definition of the instructions. However, execution speed suffers because each identifier access is preceded by a case distinction. This is a first hint that the primary design focus of the abstract code is not execution speed but compact definition and ease of (just-in-time) compilation.

(1) Allocation

Allocation in the abstract code means creation plus initialization. The style is the same for all values: first the destination is given, then several data elements for initialization are provided, and the last component `next` identifies the successor instruction. Figure 3.5 shows allocation instructions for the restricted value set.

instr	::=	...				
|	`NewTup`	{`dest` : *def*	`fields` : *ref**	`next` : *instr*}		
|	`NewTagVal`	{`dest` : *def*	`tag` : *int*	`fields` : *ref**	`next` : *instr*}	
|	`NewClosure`	{`dest` : *def*	`globals` : *ref**	`code` : *code*	`next` : *instr*}	

Figure 3.5: instructions for allocation

(2) Access

To select components from a tuple, the abstract code offers the instruction `SelTup` (Figure 3.6). The number of destination locations is equal to the tuple arity, which means that all fields are selected at once. `Wildcard` is used for all fields that should not

instr	::=	...			
|	`Let`	{`dest` : *def*	`src` : *ref*	`next` : *instr*}	
|	`SelTup`	{`dest` : *def**	`tuple` : *ref*	`next` : *instr*}	

Figure 3.6: instructions for access

be selected. The `Let` instruction loads immediate or global values to local variables, or it assigns the content of one local variable to another. As indicated before, this is used to explicitly synchronize variables after control flow join points.

(3) Conditionals

The two test instructions, depicted in Figure 3.7, work on tagged values. Both instructions subsume four basic steps:

1. They extract the tag t from the test value.
2. They look up the tag in a vector of alternatives.
3. The corresponding entry in the vector contains a list of local identifiers and a successor instruction s. The local identifiers are bound to the fields of the tagged value.
4. The successor instruction s is taken.

The general test and the compact test only differ in the second step. In the general case, the vector contains triples of the form $(tag \times def^* \times instr)$. A linear search on the vector is performed to find an entry with tag t. If there is no corresponding entry, the `else` branch is taken.

Compact tests are used if the test cases cover a contiguous range of tags from 0 to k. Then, the branches can be stored compactly in a vector, such that their position defines the associated integer tag. After a range check $0 \leq t \leq k$, the tag t serves as an index into the vector, yielding constant time lookup. The range check can be removed if the test is exhaustive, which means that the test table covers all possible cases. The static Alice compiler can find out about exhaustive tests on the basis of type information. For all exhaustive tests, `else` branch is set to NONE.

```
instr  ::=              ...
  |    TestTag          {
                          testval : ref
                          branches : (int × def* × instr)*
                          else : instr
                        }
  |    TestTagCompact   {
                          testval : ref
                          branches : (def* × instr)*
                          else : instr option
                        }
```

Figure 3.7: instructions for testing tagged values

(4) Procedure Application

To execute an Alice procedure, one has to apply actual arguments to the procedure. Two kinds of `apply` instructions are distinguished. If the caller needs the result of the callee's computation in order to resume its own computation, the application defines a *continuation*. This is a pair of a list of identifiers that store the return values of the callee, and the successor instruction of the application. In this standard form of procedure application, the new task is created on top of the caller task. The situation is different if the caller just returns the callee's return values without computing something in between. One says that the call instruction is in tail position. In this case, there is no continuation needed and the tail call instruction discards the caller task before it creates the new task.

When the callee finishes it computation, it passes some values to its caller with the `Return` instruction. The described instructions are depicted in Figure 3.8.

(5) Exceptions

Figure 3.9 shows the instructions to raise and handle an exception. `Try` pushes a new handler onto the handler stack and tries to execute the body in which an exception might occur. `EndTry` pops the topmost handler from the handler stack. An exception value is raise with `Raise`.

The handler procedure is defined by the `handler` field in the `Try` instruction. This contains the formal argument and a reference to the handler code, which is a subgraph of the normal abstract code graph.

```
instr  ::=          ...
  |    Apply        {
                     closure : ref
                     in : ref*
                     cont : (def* × instr)
                    }
  |    ApplyTail   {closure : ref  in : ref*}
  |    Return      {values : ref*}
```

Figure 3.8: instructions for procedure application and return

```
instr  ::=       ...
  |    Try      {body : instr  handler : (def × instr)}
  |    EndTry   {next : instr}
  |    Raise    {exception : ref}
```

Figure 3.9: instructions to raise and handle exceptions

Join Points

A node in the abstract code graph is a *join point* if it has two predecessors. The abstract code explicitly encodes join points as special nodes. So by definition, all other nodes (except the root node) have exactly one predecessor.

```
instr  ::=       ...
  |    Shared   {stamp : int   next : instr}
```

The `Shared` instruction is just a marker for a join point. It contains a stamp that distinguishes it from other join points in the code graph.

3.1.3 Primitives

The presented groups of abstract code instructions do not provide the full functionality of the Alice system. There is, for instance, no instruction to initiate concurrent computation and the instruction set does not contain arithmetic instructions and operations on arrays or strings. The language layer implements these operations as *primitives*, i.e. as language-specific virtual machine services. Primitives are encoded as procedures that can be used in `Apply` and `ApplyTail` instructions.

Outsourcing of core functionality to virtual machine services allows a compact definition of the abstract code. However, procedure application always causes overhead because a new task is created and the old task must possibly be kept. The effort only pays off if the primitive is sufficiently complex. An optimizing execution unit has to convert frequently used primitives into a more efficient representation. For the native code system, this means that the compiler has to detect, for instance, primitives for arithmetic and generate native code for them. In the byte code system, the interpreter offers arithmetic instructions that the compiler uses in code generation.

3.1.4 Discussion

Java uses a stack-based byte code [30] as platform-independent representation. Efficient Java virtual machines [1, 25] have to extract the control flow from the linear byte code before a run-time compiler can generate more efficient code. The Alice Abstract Code has a significant advantage over the linear byte code representation. The graph explicitly encodes the control flow, and the nodes (instructions) carry all information that is needed to directly compile the code. Therefore, a compiler does not need to construct an extra intermediate representation.

Another advantage is that code can be stored in a compact way due to the use of high-level instructions and primitives. Code size becomes especially important when code is transferred over a network connection.

On the other hand, the removal of ϕ-functions in the abstract code might artificially extend the liveness of a variable across branches. Suppose a left branch sets a variable at the beginning and the right branch synchronizes this variable at the end. Then, the variable is declared to be live over the whole right branch. This increases register pressure and decreases code quality for register-poor architectures.

3.2 Native Code

Before this thesis project, just-in-time compilation from abstract code to native code was the only way to speed up execution. In principle, the native code compiler is not biased towards a special hardware architecture. It uses the *GNU lightning* library [12] for dynamic code generation.

GNU Lightning

GNU lightning offers a meta instruction set that resembles a RISC architecture. The interface supports a wide range of current processor architecture including all 32-bit x86 architectures, the PowerPC, and SPARC. The compiler writer has to generate meta-code and *GNU lightning* directly emits machine code from the meta-code instructions. This translation process is based on macros and there is no intermediate compiler data structure involved. This makes code generation fast and does not consume much memory.

The just-in-time compiler of Alice works for all 32-bit x86 architectures. Unfortunately, porting it to PowerPC failed. The compiler uses some machine-dependent quirks that only work for the x86 architecture. It is not enough to recompile it with different settings for *GNU lightning* – this is the intention of the library. Instead, several parts of the just-in-time compiler have to be rewritten. To make thing worse, *GNU lightning* currently does not seem to be in a stable state for PowerPC.

Drawbacks of GNU Lightning

GNU lightning is one of the best tools for dynamic code generation that is currently available under a free software license. Nevertheless, the library has some severe limitations.

Implementing a (just-in-time) compiler is an error-prone task and decent debugging facilities are crucial to succeed in acceptable time. It is the most critical drawback of *GNU lightning* that it does not offer a symbolic debugger. This means that the

abstraction layer, i.e. the meta instructions, can be used to write programs, but there is no facility to read the program on this meta level. In order to debug the created code, one has to step through the actual machine code, which requires deep knowledge of the underlying hardware.

The compiler writer can only access a limited number of registers. Currently the number of registers is six. The Alice just-in-time compiler needs three registers for temporary use. This leaves only three registers for register allocation. So the design of a meta instruction set for both RISC and CISC architectures led to the least common denominator with respect to register numbers. Of course, this saves a possibly complex mapping from virtual to real registers, but it impedes effective usage of registers on register-rich architectures. Additionally, the library does neither provide a peephole optimizer nor an instruction scheduler. Both techniques would increase performance, but, as the designers say, would considerably slow down code generation.

To sum up, it is quite involved to generate native code. The developer of a native code compiler needs deep knowledge of the underlying hardware to generate efficient code. Using a library such as *GNU lightning* eases implementation. However, the additional abstraction layer has some drawbacks since it does not feature decent debugging facilities and since it does neither apply local nor global code optimizations.

3.3 Motivation for a Byte Code System

To generate native code of acceptable quality, many of the abstractions that Seam provides have to be reimplemented in assembler code, so that the just-in-time compiler can inline them. It would not be feasible to use function calls for all abstractions. The assembler programming part highly increases the engineering effort and undermines the design idea of Seam, which is that the language implementor can just use generic services without caring about their internal realization.

This project takes a different approach with the goal of achieving efficient execution as well as portability. The key idea is to use *byte code* instead of native code, and to generate the byte code at run-time from abstract code. To execute the byte code, an interpreter is implemented that respects Seam's interfaces. The great advantage is that the whole engine can be implemented in a high-level programming language (here: C++) and no knowledge of the target machine is required. All Seam abstractions can be used in the interpreter implementation. When the interpreter is compiled, modern C++ compilers, like *GCC*, automatically inline most of the abstractions. Thus, using the abstractions does not cause performance penalties.

A highly optimized native code system will undoubtedly always be faster than a byte code interpreter. However, for a research system like Alice, the second approach has significant advantages. The main focus of Alice is the investigation of high-level programming features. The underlying virtual machine has to be maintainable, easy to extend, independent from the hardware architecture, and reasonably fast. A byte code system suits these demands best, especially when the developer team is small. Additionally, such a system is an ideal research platform to investigate high-level optimizations because it is much easier to debug than a native code compiler. The knowledge that the developers gain from the byte code system helps a lot to identify the strengths and weaknesses of the whole programming system. In fact, it will be useful even if the team finally decides to move the research focus to the underlying virtual machine and develop a sophisticated native code compiler.

4 Byte Code Interpreter and Instruction Set

This chapter develops a byte code interpreter for the Alice virtual machine. The first section defines what is meant by *byte code* in this thesis. Thereafter, the concrete machine model of the byte code interpreter is explained. This includes the principle architecture and the definition of a byte code task. The third section specifies the byte code instruction set. We conclude the chapter by introducing some optimizations that are applied in order to achieve good performance.

4.1 Definition

The term *byte code* is often used and everybody has a rough idea what it means. However, there is no official definition for the term and it is used for quite different code forms. Therefore, we give a definition for what *byte code* means in the context of this thesis.

Definition (Byte Code) *Byte code is a sequential representation of a program, consisting of virtual machine instructions that may have scalar arguments (here: integers). The virtual machine instructions can be executed by an interpreter.*

So byte code instructions are executed by a software machine and not directly on hardware. We use the term *byte code buffer* to describe the contiguous memory region in which the byte code instructions and arguments are aligned in a string-like fashion. Instructions and arguments are not necessarily byte-aligned.

4.2 Machine Model

Byte code tasks are executed by the byte code interpreter that implements a register-based machine model. Figure 4.1 depicts a byte code task with all state information.

Registers

All local variables of a procedure are kept in registers that live in the current task. The number of registers that are needed during the execution of a task can be computed at compile time. So the task creator preallocates a task large enough to hold all registers. In the following, the term *register* always refers to the location in the task. In order to clearly distinguish them from hardware registers, they are sometimes called byte code registers or virtual registers.

Machine State

In addition to the registers, the interpreter state comprises four components:

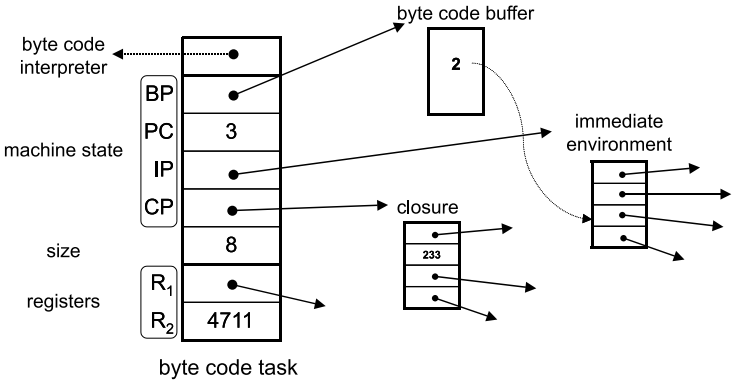


Figure 4.1: byte code task and state information

BP	pointer to byte code buffer
PC	program counter
IP	pointer to immediate environment
CP	pointer to closure

Every Alice procedure maintains its own byte code buffer. The buffer is represented as a chunk and contains all byte code instructions that implement the procedure. The program counter points to the instruction that is currently executed.

The byte code instructions do not directly access immediate values that are subject to garbage collection. Instead, they use indexes into an *immediate environment* to access the values indirectly (Section 2.2.2). Whenever the interpreter loads an immediate value into a register, it performs the assignment $R_i \leftarrow IP[j]$.

Every procedure maintains an environment to access global variables. CP points to the closure that keeps the values for all global variables that are used in the current byte code chunk. Global variables are indexes into the closure. So to load, for instance, the content of global variable number 2 into register R_4, the interpreter executes the assignment $R_4 \leftarrow CP[2]$.

Byte Code Task and Startup

A byte code task contains all registers and a complete description of the interpreter state *(BP,PC,IP,CP)*. Since the number of registers varies from task to task, it additionally maintains a size information. Figure 4.1 shows the layout of a byte code task. Behind the reference to the interpreter, which is common to all tasks in SEAM, the state and the size is defined. The registers lie behind the size information.

The scheduler starts the interpreter by passing a byte code task to it. The interpreter fetches the state information and starts execution at $BP[PC]$.

Creating a byte code task involves compilation from abstract code to byte code. The compiler is explained in the next chapter.

Interrupts

There are three situations in which the interpreter has to interrupt its current computation and return control to the scheduler.

- *Transients:* When the interpreter requires the value of a transient, it issues a request to the scheduler that coordinates evaluation and synchronization.
- *Preemption:* The interpreter periodically checks the global status register and returns control to the scheduler if the status register is set. This happens, for instance, if a garbage collection becomes necessary or if the scheduler chooses another thread to resume its computation.
- *Exceptions:* As the scheduler is in charge of exception handling, the interpreter returns control whenever an exception is raised.

When an interrupt occurs, the interpreter stores its state inside the current task. The program counter is the only information that has to be saved since the three remaining state components do not change during execution, and registers are automatically caller-saved since they are local variables of the caller task.

Exceptions

The byte code system represents a handler procedure as a pair (x, pc). The byte code register x is the binder for the exception value. The handler code starts at $BP[pc]$. The task creator for handler procedures assigns the content of the scheduler register R_{exn} to x and sets the PC of the current task to pc.

Procedure Calls

The easiest way to implement procedure calls is to follow SEAM's standard protocol. This means that the interpreter transfers the arguments to the scheduler registers and issues a procedure call to the scheduler. The scheduler initiates task creation and returns control to the interpreter again. This scheme works perfectly, but is not efficient in practice. We will see in Section 4.4 how to optimize procedure calls by bypassing the scheduler.

Related Work

There is no consensus that register-based interpreters are superior to stack-based architectures. Historically, both approaches have coexisted with a preference for stack-based interpreters. Two early virtual machine implementations, using stack-based interpreters, are the Algol Object Code (1964) [35] and the P-machine (1976) [5] for the execution of Pascal programs. The notion of registers in virtual machine occurred in Warren's Abstract Machine (1983) [2].

The virtual machine of OCaml is an example for an efficient stack-based interpreter. The standard stack architecture is slightly modified in order to keep the topmost stack cell always in a register. The idea comes from Leroy's "The ZINC Experiment" [29]. Ertl proposed a more general approach to cache a variable amount of stack values in registers [18].

However, during the last years there has been a trend towards register machines. Shi et al. transform Java byte code to a register format at load-time and report about a speedup of 26.5% on average [41]. The reason is that the number of executed instructions drops more than 47% when a register instruction set is used. However, code size increases by 25% since the arguments have to be specified explicitly.

The designers of Lua 5.0 switched from a stack to a register architecture and report a speedup of 18% on average for their benchmarks [24]. The register design of the Alice byte code interpreter closely resembles Lua 5.0. One difference is that, for technical

reasons[1], Lua imposes a limit of 200 registers per stack frame. In contrast, the size of a byte code task is only limited by the available hardware memory. Due to the SSA form of the abstract code, there are sometimes more than 3000 registers needed. Since this results in huge tasks, Section 6.4 introduces a form of register allocation that reduces the register needs.

4.3 Byte Code

The abstract code is the basis for the Alice byte code, or byte code for short. The idea is to remove all information that is not needed for the computation, and to flatten the abstract code graph. So the byte code is mainly a linearization and specialization of the abstract code.

The first section describes the structure and the layout of the byte code instructions. After that, the instruction set is introduced. As for the abstract code, we do not present all instructions, but focus on the restricted value set (integers, tagged values, closures, tuples, and exceptions).

4.3.1 Structure and Layout

A byte code instruction consists of an opcode, possibly followed by three types of arguments. These are register numbers, immediate values and jump offsets, which are all of type integer. The register numbers specify locations in the current byte code task. Immediate integer values are directly stored in the code, whereas immediate values that are subject to garbage collection are stored in an immediate environment. The code stores the indexes to this environment, instead of the actual values. Jump instructions define their target by an offset that is added to the current program counter:

$$\mathit{target} = PC + \mathit{offset}$$

Given a set of opcodes, which are operations like `iadd`, and the symbols c, r, v, and o with the following meaning

$c \in \text{Opcode}$	opcode
$r \in \mathbb{N}_{\geq 0}$	register number
$v \in \mathbb{N}$	value (integers or address into immediate environment)
$o \in \mathbb{N}$	jump offsets

the abstract layout of all byte code instructions can be defined as a regular expression

$$\mathit{instr} \stackrel{\text{def}}{=} c\, r^*\, v^*\, o^*$$

The abstract instruction layout specifies two important properties. First, the order of arguments of different type is fixed, and second, an instruction can have an arbitrary number of arguments. The actual layout that is used in the implementation is given in Chapter 7, which details implementation specific aspects of the byte code system.

[1] Lua allocates the stack frames in the C stack. The restriction is an arbitrary choice to limit the stack growth on function calls. Compilation aborts if the limit is exceeded.

As mentioned before, the abstract code does not contain polymorphic instructions. This property extends to byte code instructions. So every instruction knows how to interpret its arguments. There will never be confusion whether the integer is a constant, an index into the global environment, or an index to the immediate environment.

4.3.2 Instructions

This section presents all instructions that correspond to the abstract code instructions from Section 3.1.

Identifiers

The byte code instructions use registers for all variable arguments. This way, the tests that the abstract code instructions perform on applied variable occurrences are removed.

Registers correspond to local variables and thus there is no instruction needed to load local identifiers of the abstract code to byte code registers. The byte code compiler is in charge of mapping the local identifiers to byte code registers. The simplest mapping is the identity function $Local(i) \mapsto R_i$.

Global variables and immediate values are explicitly loaded into registers to make them accessible to the instructions.

```
load_global     dst  addr
load_immediate  dst  addr
load_int        dst  number
```

For global variables, `addr` is the index into the closure, and for immediate values, it is the index into the immediate environment. Notice that the integer values are not stored in the immediate environment because they can be embedded into the code. `load_int` moves the integer `number` into the destination register.

(1) Allocation

The abstract code instructions combine construction and initialization. These phases are separated in the byte code instruction set. There is one instruction to construct a new value and to store it into a register, and there is another instruction to initialize the fields one by one.

- Allocation of tuples:

  ```
  new_tup   dst  width
  init_tup  dst  src    index
  ```

  The first instructions creates a new tuple of arity `width` and stores the result in register `dst`. The second instruction takes the fresh tuple, given by `dst`, and copies the content of register `src` into field number `index`. For instance, `NewTup IdDef(1) [Local(2), Local(3)]` corresponds to the byte code sequence `[new_tup R1 2, init_tup R1 R2 0, init_tup R1 R3 1]`.

- Allocation of tagged values:

  ```
  new_tagval   dst  size  tag
  init_tagval  dst  src   index
  ```

The instruction `new_tagval` creates a new tagged value with `size` fields and the identification number `tag`. Initialization is identical to tuples.

- Allocation of closures:

```
mk_closure    dst  code  size
init_closure  dst  src   index
```

`mk_closure` creates a new closure into register `dst`. `code` is an index into the immediate environment in which the reference to the code is stored. The environment of the closure has `size` fields that can be filled one by one with `init_closure`.

(2) Data Structure Access

Once values are allocated, they can be moved to other locations, for instance, to synchronize variables of two branches (cf. SSA form, Section 3.1.1). The semantics of the load instruction is that the content of register `src` is copied to register `dst`.

```
load_reg  dst  src
```

Another operation on data structures is selection. The following byte code instructions select a single field from a tuple or a tagged value:

```
select_tup   dst  src  index
load_tagval  dst  src  index
```

`src` specifies the tuple or tagged value, `index` the field, and `dst` the register in which the field should be stored.

All instructions that require the actual value of an argument contain an internal test on transient values. If the test finds out that the argument is not yet determined, the interpreter interrupts execution and issues a request to the scheduler. There is, however, a single situation in which we need an explicit transient test: pattern matching on the empty tuple, i.e. the unit value. The static compiler generates an empty tuple selection `SelTup (#[],Local(0))` from source code of the form `fun xyz () = ...`. This corresponds to an explicit request, as the semantics of Alice prescribes that a value has to be determined before it is selected. The byte code offers an extra instruction for this case that checks for a transient value and interrupts execution if the value is not yet determined.

```
await src
```

(3) Conditionals

The byte code distinguished three different types of tag tests: (a) general tests, (b) compact tests, and (c) exhaustive tests.

```
tagtest          R0  table                    (* (a) *)
ctagtest         R0  (* inlined table *)      (* (b) *)
ctagtest_direct  R0  (* inlined table *)      (* (c) *)
```

All three instructions keep the test value in register `R0`. The second argument specifies the test table. On execution, the instructions first extract the tag out of the test value and look up the tag in the test table, afterwards.

`tagtest` implements the test table as hash table, and `table` is its address in the immediate environment. The tag test checks if the extracted tag is a member of the hash table and sets the program counter to the continuation address that is stored in the table.

The instructions do not contain explicit information about the `else` branch. If there is one, the compiler has to assure that the code for the `else` branch is directly emitted behind the tag test instruction.

The instruction `ctagtest` is for compact tests, whereas `ctagtest_direct` can be used for exhaustive tests. For both instructions, the test table is inlined into the code. Figure 4.2 exemplifies the layout of a compact tag test with an inlined test table. The

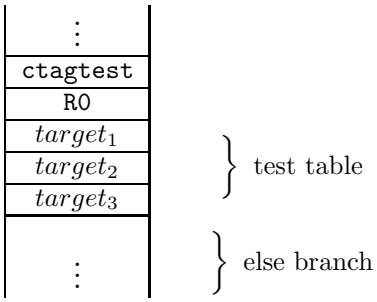


Figure 4.2: tag test with inlined test table

difference between the compact test and the exhaustive test is that the latter does not perform a range check. This implies that exhaustive tests do not have an `else` branch.

(4) Procedure Application

The instruction set provides a call instruction that respects the standard SEAM protocol. There is an additional instruction to treat tail calls more efficiently.

```
seam_call      R0  R1 ... Rn
seam_tailcall  R0  R1 ... Rn
```

Register `R0` contains a closure, and registers `R1` to `Rn` contain the arguments. The instruction transfers all arguments to scheduler registers and issues a procedure call to the scheduler.

The return instruction finishes the execution of a task. It writes the contents of registers `R1` to `Rn` to the global register bank and removes the current byte code task from the task stack. Then it issues the scheduler to resume execution of the underlying task.

```
seam_return  R1 ... Rn
```

(5) Exceptions

To implement exception handling, there are byte code instructions to push and pop an exception handler, and there is an instruction to raise an exception:

```
install_handler  h
remove_handler
raise            R0
```

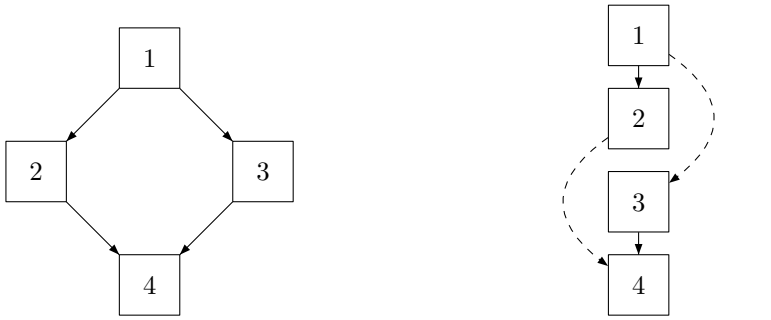
`install_handler` pushes the handler, which is stored under index `h` in the immediate environment, onto the handler stack. The handler is popped from the stack with the `remove_handler` instruction. The instruction `raise R0` raises the exception value that is stored in register `R0`.

Join Points

In order to encode the control flow graph in a linear byte code representation, the byte code simulates some of the graph edges with jump instructions. The jump target is defined relatively to the jump instruction. To execute the jump, the interpreter adds the offset to the current program counter.

```
jump offset
```

Figure 4.3 depicts a control flow graph together with its linear representation.



(a) control flow graph

(b) linear representation with jumps

Figure 4.3: general control flow graph and its linear representation

The control flow splits after node 1, for instance, because of a two-sided tag test. Behind node 2 and 3, the control flow joins again in node 4. The first step to obtain a linear representation is to store all nodes of the graph in an array. As the linear representation does not permit explicit edges, i.e. references from one node to another, they have to be simulated.

The execution path of the byte code interpreter follows the linear order of the instructions that are stored in the array. So the array representation implicitly defines edges $(1, 2)$, $(2, 3)$, and $(3, 4)$. As we are interested in an isomorphic representation, we have to convert the edge $(2, 3)$ into an edge $(2, 4)$. Therefore, a `jump` is inserted at node 2 to skip node 3. The edge $(1, 3)$ results from the tag test in node 1.

Calling Convention Conversion

There must be some way to load procedure arguments out of the global register bank into byte code registers. Due to the calling convention of Alice (Section 2.2.3), the number of arguments that are in the register bank does not necessarily coincide with the receiver's expectation. So in some cases, the receiver has to convert the arguments before loading them into byte code registers. The two CCC instructions in the byte code perform the conversion internally and then load the arguments into the dedicated registers.

```
ccc1  R1
cccn  R1 ... Rk  (* k > 1 *)
```

The first instruction indicates that the receiver expects one argument and that the argument will be loaded into register `R1`. If the instruction finds more than one argument in the register bank, it constructs a tuple; and if there is no argument, it stores the unit value into `R1`. The second byte code instruction checks if there are several arguments present. If there is only one, this argument is certain to be a tuple that is deconstructed into registers `R1`,...,`Rk`. Otherwise, the arguments are loaded into `R1`,...,`Rk`.

4.4 Optimizations

The presented byte code instructions suffice to express every Alice program that is defined on the restricted value set. However, the resulting program would be slow. In order to offer more efficient instructions for frequent special cases, the base instruction set is extended with specialized instructions. Furthermore, there are specialized call instructions that bypass the scheduler whenever possible.

Inlined Primitives

Primitives implement core functionality of the language as virtual machine services. SEAM defines a task creator and an interpreter for primitives. This way, they can be used as normal procedures. However, if the primitive is small and frequently used, execution speed significantly suffers from the overhead of many procedure calls. The byte code interpreter re-implements a small set of primitives. This contains instructions for integer arithmetic, operations on reference cells, and an instruction to create by-need transients.

```
iadd                  dst   src1  src2
isub                  dst   src1  src2
set_cell              cell  src         (* cell := src *)
load_cell             dst   src         (* dst = !src *)
inlined_new_byneed    dst   src
```

Dynamic Tests for Intra-Language Invocation

The interpreter can significantly speed up procedure calls if it bypasses the scheduler. Most of the procedure calls are so called *intra-language invocations*, i.e. the current interpreter also executes the procedure that is invoked. To leverage this, the (tail)call

instruction tests for every procedure if it has a byte code representation. In this case, it does not use the scheduler to invoke the procedure. Instead, the instruction itself creates the new task, updates the interpreter state, and starts execution at the new program counter. If the interpreter detects a recursive call, it can even perform better since it only needs to create the new task and update the program counter.

Dynamic tests in the call instructions prove to be quite effective in practice. They produce low overhead and significantly speed up execution. The great advantage of dynamic tests over static optimization is that they really catch *all* byte code invocations. This is especially interesting in Alice where procedures can be passed as arguments. The feature is heavily used, for instance, in `foldl` for lists.

```
fun foldl _ a  nil     = a
  | foldl f a (x::xr) = foldl f (f(x,a)) xr
fun my_op (x,a) = 3*x + a

... foldl my_op 0 xs ...
```

The example shows the implementation of `foldl` and an operator `my_op`. The custom operator `my_op` is applied to `foldl` somewhere in the program text. Suppose that both procedures are automatically translated to byte code on their first invocation. Clearly, all calls in `foldl` to the operator are byte code calls and the dynamic call tests ensure that they are efficiently invoked. Static optimization cannot do anything because the call inside `foldl` is to an argument that is likely to change during run-time.

Whenever the interpreter bypasses the scheduler, it is in charge of simulating all essential scheduler operations. The interpreter executes the following steps for a byte code call:

1. It calls the task creator to construct the new task.
2. It tests for preemption and possibly returns control to the scheduler.
3. Otherwise, it adjusts interpreter state to the new code, closure and immediate environment, and sets the program counter to the beginning of the new code.

Parameter passing does not change. The scheduler registers still serve as intermediate locations for the arguments.

Dynamic tests are also used in the return instruction. Instead of giving back control to the scheduler, the interpreter tests if the task to which it returns is a byte code task. In this case, it can directly execute it.

Dynamic Code Rewriting

In a high performance interpreter, the dynamic test should be avoided whenever possible. The instruction set offers special calls for immediate procedures. The call instructions are separated into three classes:

- *Byte code call:* This instruction assumes that the callee has a byte code representation and it bypasses the scheduler without a dynamic tests.
- *Rewrite call:* This instruction dynamically tests for byte code calls. If it detects such a call, it rewrites itself to a byte code call instruction. This way, there are no tests in the following executions.
- *Immediate call:* This instruction can be used if it is clear that the procedure does not have a byte code representation. It issues a call to the scheduler without performing a dynamic test.

Rewrite call instructions are especially interesting. For many immediate procedures, it is true that they may be transformed to byte code in the future. For instance, the byte code system compiles a procedure when it is executed for the n-th time. Dynamic code rewriting means to overwrite the old instruction with a new one. The byte code interpreter replaces the opcode, but keeps the arguments. Rewriting becomes a local operation that does not require further adjustments on the code, like recomputation of jump offsets.

The native code compiler also optimizes procedure calls. These optimizations strongly depend on the assumption that procedures are compiled on their first use. This means that postponing compilation further breaks the call optimizations.

Instruction Specialization

A concise set of byte code instructions is nice for presentation and it eases compiler implementation. However, for many instructions, there are special cases that occur quite often. Adding specialized instructions to the instruction set noticeably increases execution speed.

Alice offers tuples of fixed arity. In practice, most of the tuples are pairs and triples. Therefore, it pays off to introduce specialized byte code instructions for construction and deconstruction of pairs and triples. Other examples for specializations are call instructions. The number of arguments rarely exceeds three, such that performance benefits from specialized call instructions for argument numbers from zero[2] to three arguments. There are also specialized arithmetic instructions, for example, an increment and a decrement instruction for integer arguments. The semantics of Alice prescribes overflow and underflow checks for all arithmetic operations. The general instruction to add two integers checks on the result whether it falls below the smallest representable number or exceeds the biggest representable number. The increment instruction only needs to check for overflow, and the decrement instruction only needs to check for underflow.

There are two main reasons for the performance increase. First, the specialized byte code instruction can be implemented more efficiently. Second, the dispatch overhead for arguments is lower. For instance, the increment instruction only fetches one register number from the code, whereas the general `iadd` instruction requires access to three fields (one target register and two source registers).

Super-Instructions

Super-instructions combine several virtual machine instructions into one, thus reducing code size as well as dispatch and argument access overhead. The literature distinguishes two ways of implementing super-instructions, namely a *static* and a *dynamic* approach [19]. In the static approach the interpreter writer identifies common instruction sequences and adds a new implementation for each super-instruction. In the dynamic approach super-instructions are created at run-time.

The Alice byte code system uses static super-instructions. There is, for instance, an instruction that combines pair construction and initialization.

```
new_pair R2
init_tup R2 R0      ==>    new_pair_init R2 R0 R1
init_tup R2 R1
```

[2] In Alice, the caller does not pass the unit value. If the callee needs the argument, the calling convention conversion at procedure entry takes care to supply unit.

The super-instruction `new_pair_init` combines the three instructions into one. Assuming that for every instruction and every argument one code buffer slot is needed, code size decreases from eight slots to four slots. Instead of three dispatch operations, there is only one, and argument `R2` is also accessed once instead of three times.

While implementing static super-instructions is relatively straightforward, implementing dynamic super-instructions is more demanding. Piumatra and Riccardi propose a technique that they call *selective inlining* [33]. The idea is to combine all instructions in a basic block by concatenating their implementations at run-time. On some architectures, a few assembler code instructions have to be interspersed between the old implementations. So it is not possible to implement dynamic super-instructions in a portable way. As one of the main goals of this project is to achieve platform independence and portability, the dynamic approach is not appropriate for the byte code system.

Related Work

The main trend to speed up execution in virtual machines is compilation to native code at run-time. Several variations of such systems exists. The programming language *Self* is executed on a *compile-only* system [23]. Sun's Java hot spot virtual machine [1] only compiles *hot paths* and thus can afford to apply expensive optimizations that are common to static compilers.

Nevertheless, interpreters are an important part of modern virtual machines. As hotspot compilers only transform a part of the byte code to native code, the rest is executed by an efficient interpreter. Some important techniques to optimize the dispatch of the next instruction can explained best at source code level. They are discussed in the implementation chapter (Section 7.2).

5 Byte Code Compiler

This chapter deals with compilation from abstract code to byte code. Byte code is generated at run-time. This is called *just-in-time* or *dynamic* compilation. The first section discusses why just-in-time compilation is particularly well-suited for Alice. The compilation process is then explained by means of a translation function that traverses an abstract code graph and emits semantically equivalent byte code. Several infrastructure components that are needed in the specification of the translation function are introduced before. Translation function and infrastructure form the compiler, which is specified in the last section.

5.1 Why Just-In-Time?

Aycock [8] characterizes just-in-time (JIT) compilation in the following way:

> Strictly speaking, JIT compilation systems are completely unnecessary. They are only a means to improve the time and space efficiency of a program.

The central problem of executing Alice programs is solved by static compilation to abstract code, which is interpreted at run-time. However, to speed up execution, just-in-time compilation is especially well-suited for Alice.

In Alice, procedures are grouped together in *components* that are the physical correspondence of modules (Rossberg et al. [38]). The static Alice compiler compiles each component separately to abstract code. A component can access other components by explicitly declaring imports. An import path is given as a URL, either stating a local path or a general location in the Internet. All components are linked at run-time when the functionality that they provide is needed for the first time.

Separate compilation and the dynamic features of the component system make it difficult or even impossible for the static compiler to do cross-component optimization. This means, for instance, that the static compiler cannot optimize primitive calls. The primitives are defined in components (for example `Int.+` in structure `Int`) and the compiler cannot differentiate between primitives and user-supplied procedures. The only way of cross-components optimization is to bundle several components that are locally available. This so called *static linking* cannot heavily be used as arbitrary growth of components is not desirable. Currently, the static linker bundles precompiled components, but does not apply further optimizations. So statically, there is no cross-component optimization at all.

At run-time, there is much more information available about the current state of the system. In particular, the encapsulation of procedures into components is completely resolved. The run-time system only sees a uniform collection of procedures. A just-in-time compiler can apply powerful techniques like *inlining*, which means to replace the call site of a procedure by its implementation. This way, the compiler can generate

more efficient code for primitives, for instance `iadd` instead of `op+`, or it can embed the body of a normal procedure into the call site.

It has to be mentioned that the just-in-time compiler knows more about the system state than the static compiler, but is does not know everything. Suppose the compiler starts at time t. At this time, the run-time system already loaded and linked a collection $C_{\leq t}$ of procedures. The run-time compiler can access all procedure in $C_{\leq t}$ to generate optimized code. However, it cannot access procedures that will be loaded at time $t' \geq t$.

Trade-Off

Static compilers can use expensive optimizations because the overhead does not slow down the run-time system. Dynamic compilers have to be much more efficient in both time and space because the compilation overhead directly affects the overall execution time of the system.

Compilation from abstract code to byte code is fast since the byte code instructions closely resemble the abstract code instructions and there is no intermediate representation needed. However, the simple compilation scheme presented in this chapter does not produce efficient code. The next chapter introduces several optimizations and shows how they can be realized efficiently.

Related Work

Aycock gives an overview about "The History of Just-In-Time" [8]. Just-in-time compilation is an old technique that has become popular again with the advent of Java [21]. Platform independence, type safety, and features for web programming have made Java interesting for commercial use. Initially, Java systems were really slow because they interpreted statically created byte code. Static compilation to native code cannot support dynamic class loading; so a just-in-time compiler is more appropriate since it can apply more optimizations [1, 25].

5.2 Infrastructure

The main component of the compiler is the translation function that transforms abstract code into byte code. This section describes several services that the compiler provides to prepare and support the translation process.

Pseudo Code Conventions

All components of the compiler are formulated as procedures in *ML*-like notation. There are three differences to the real programming language:

- *Delimiters:* Instead of `EMIT("a b c")`, we write `EMIT (a b c)`. Additionally, we omit some parentheses and use spaces instead.
- *Polymorphic Maps:* We assume that there is a universal hash table *Map* that works for every atomic type.
- *Variable number of arguments:* In some cases, variable length patterns are used.

```
val [x1,...,xn] = map f xs   (*definition*)
val y = h x1 ... xn          (*use*)
```

In the example, the new list is deconstructed and the arguments are applied one after another to procedure `h`. This notation saves explicit formulation of loops.

Additionally, the implementation of some procedures is not given when it is clear from their signature what they do.

Code Buffer

The byte code buffer, introduced in Section 4.1, is constructed during compile-time. The compiler emits the byte code into a code buffer `code` that can be thought of as an array of integers. The macro `EMIT` abstracts away the details of alignment. To write an instruction with several arguments into the code buffer, `EMIT` is used in the following way:

```
EMIT (opcode arg1 ... argn)
```

The procedure `codebuffer_init` initializes the buffer for a new compilation. The first free position in the code buffer is denoted by PC.

Immediate Environment

During code generation, the compiler sets up the immediate environment `ienv` for those values that must not be stored in the code. The following operations can manipulate the environment:

```
val ienv_insert : value -> index
val ienv_init   : unit  -> unit
val ienv_update : index*value -> unit
```

The procedure `ienv_init` resets the global immediate environment. An immediate value is registered at `ienv` with `ienv_insert`. Byte code instructions use the index to access the value. The environment can be changed during compilation with `ienv_update`. This operation can be used to emit the byte code instruction before the actual value is known.

Offset Table

The abstract code graph is a directed acyclic graph with explicit join points. The byte code simulates the control flow edges with jump instructions. The compiler uses an *offset table* to compute the offsets of the jumps. When it visits a join point for the first time, it compiles the successor instructions and invokes `set_visited` to store the start address of the corresponding byte code instructions in the offset table. When it visits the join point again (coming from the other branch), it computes the offset between the current PC and the start address in the table with `get_offset` and inserts a jump instruction.

```
val set_visited       : join_point_stamp -> unit
val get_offset        : join_point_stamp -> int
val visited           : join_point_stamp -> bool
val offsettable_init  : unit             -> unit
```

`offsettable_init` resets the offset table for a compilation. With `visited`, the compiler checks if it has visited the join point before.

Register Handling

The compiler distinguishes two kinds of registers:

- *normal registers* that correspond to local identifiers in the abstract code, and
- *scratch registers* that are temporary locations that the compiler creates during compilation. Scratch registers are, for instance, used to make global or immediate values accessible to byte code instructions that only operate on registers.

Both kinds of registers are represented as locations in the current byte code task. Normal registers take up the first n locations, and scratch registers start at location $n + 1$. The compiler defines the following functor to handle registers in the described way:

```
type id = int
type register = int
functor Registers (val numberOfRegs : int
                   val mapping      : id -> register) :
  sig
    val reset  : unit -> unit
    val peak   : unit -> register
    val new    : unit -> register
    val fromId : id   -> register
  end
```

The functor is parametrized by the number of normal registers and a mapping from local identifiers to registers. The first argument specifies a lower bound for the register needs of the procedure that is to compile. The second argument specifies how the local identifiers from the abstract code are mapped to registers. The simplest solution is to use the identity function and the statically known number of local identifiers.

```
structure Registers = Registers(val numberOfRegs = numberOfIds
                                fun mapping id = id)
```

The compiler obtains a free scratch register by invoking `new`. The liveness of scratch registers is restricted to the compilation of one abstract code instruction. So all scratch registers can be freed afterwards with `reset`. At the end of the compilation process, the procedure `peak` returns the upper bound of the number of needed registers. This number defines the size of the byte code task.

Identifiers

The abstract code explicitly distinguishes between defining and applied occurrences of an identifier. Defining occurrences can either be a wildcard or a local identifier. The procedure `getIdDef` maps defining occurrences in the abstract code to byte code registers.

```
fun getIdDef Wildcard   = Registers.new()
  | getIdDef (IdDef id) = Registers.fromId id
```

It uses `fromId` to map local identifiers to virtual registers and returns a dummy register for wildcards.

Applied occurrences of identifiers are also mapped to byte code registers. In the abstract code, identifiers can be local or global variables, or immediate values. The procedure `loadIdRef` maps local identifiers to byte code registers with the mapping `Registers.fromId`. In order to load the content of global variables into a register at run-time, `loadIdRef` requests a new scratch register and emits a load instruction. For immediate values, it distinguishes between integers and pointer values. Integers can directly be embedded into the code, whereas pointer values have to be stored in the immediate environment.

```
fun loadIdRef (Local id)     = Registers.fromId id
  | loadIdRef (Immediate v) =
    if isInteger v then
       EMIT (load_int r v)
    else
       let
         val index = ienv_insert v
         val r     = Registers.new()
       in
         EMIT (load_immediate r index); r
       end
  | loadIdRef (Global id) =
    let
      val r = Registers.new()
    in
      EMIT (load_global r id); r
    end
```

The translation function uses the procedure `loadIdRef` whenever it has to load an argument into a register in order to make it available to a byte code instruction.

Calling Convention

The calling convention of Alice requires that each receiver of arguments transform the arguments into the format it expects. So the receiver checks if there is an argument mismatch, transforms the argument into the expected format and loads them out of the register bank into the specified byte code registers. The byte code provides the instructions `ccc1` and `cccn` for this purpose.

```
fun transl_ccc #[]          = ()
  | transl_ccc #[id]        =
    let
      val r = load id
    in
      EMIT (ccc1 r)
    end
  | transl_ccc formalArgs =
    let
      val [r1,...,rn] = Vector.map getIdDef formalArgs
    in
      EMIT (cccn r1 ... rn)
    end
```

If the receiver does not expect arguments, there is nothing to compile. In case it expects one argument, `ccc1` ensures tuple construction if more arguments are present. `cccn` is generated if the receiver expects more than one argument.

5.3 Translation Function

The central component of the compiler is the translation function *transl* that emits byte code for abstract code instructions. The function takes the root node of an abstract code graph and emits a sequence of byte code instructions into the code buffer `code` while traversing the graph.

```
val transl : abstract_code -> unit
```

The translation function is specified in pseudo code notation using the infrastructure introduced before. The implementation is explained separately for each abstract code

instruction, following the order in which they are introduced in Section 3.1. Scratch registers are reset with `Registers.reset` before an instruction is compiled.

Allocation: `NewTup`, `NewTagVal`, `NewClosure`

To generate code for tuple allocation, the translation function first computes the destination register and the arity of the tuple and emits the `new_tup` instruction with the corresponding arguments. In a second step, it emits one initialization instruction for every field of the tuple.

```
fun transl NewTup(dst,fields,next) =
  let
    val _     = Registers.reset()
    val t     = getIdDef dst
    val arity = Vector.length fields

    fun init (i,x) =
        let
          val r = loadIdRef x
        in
          EMIT (init_tup t r i)
        end
  in
    EMIT (new_tup t arity);  (*creation*)
    Vector.appi init fields; (*initialization*)
    transl next
  end
```

Code generation for `NewTagVal` is almost identical to tuple construction. The only difference is the additional tag. Instead of `new_tup t arity`, the compiler emits `new_tagval t arity tag`, and instead of `init_tup`, it uses `init_tagval`.

The new concept needed for closure construction is the *code constructor* that defines the internal code representation. So for instance, it tells the byte code compiler to compile the abstract code when the procedure is invoked for the first time. Everything else is compiled just like tuples. There is one instruction to create the closure at run-time and some other instructions to initialize the environment.

```
| transl NewClosure(dst,globals,abstractCode,next) =
  let
    val _     = Registers.reset()
    val r     = getIdDef dst
    val size  = Vector.length globals
    val code  = AliceLanguageLayer.codeConstructor abstractCode
    val index = ienv_insert code

    fun init (i,x) =
        let
          val r = loadIdRef x
        in
          EMIT (init_closure t r i)
        end
  in
    EMIT (mk_closure r index size); (*creation*)
    Vector.appi init fields;        (*initialization*)
    transl next
  end
```

Data Structure Access: `Let`, `SelTup`

To translate the `Let` instruction, the compiler determines the destination register and emits code depending on the source. For local identifiers, it checks whether a move

is required. Immediate values and global variables can directly be loaded into the destination register.

```
| transl Let(dst,src,next) =
  let
    val _  = Registers.reset()
    val r0 = getIdDef dst
  in
    case src of
        Local id    =>
         let val r1 = Registers.fromId id
         in if r0<>r1 then EMIT (load_reg r0 r1)
         end
      | Immediate v =>
         if isInteger v then EMIT (load_int r0 v)
         else
           let
             val index = ienv_insert v
           in
             EMIT (load_immediate r0 index)
           end
      | Global id   => EMIT (load_global r0 id);
    transl next
  end
```

Tuple selection is compiled component-wise, which means that the compiler emits one instruction per binder. A wildcard implies that the field need not be selected. If the selection operates on an empty tuple, its semantics is to request the unit value. In this case, the translation function emits an `await` instruction.

```
| transl SelTup(dests,src,next) =
  let
    val _ = Registers.reset()
    val t = loadIdRef src
    fun select (_,Wildcard) = ()
      | select (i,IdDef id) =
        let val r = Registers.fromId id
        in  EMIT (select_tup r t i)
        end
  in
    if length dests = 0 then EMIT (await t)
    else Vector.appi select dests;
    transl next
  end
```

Conditionals: `TestTag`, `TestTagCompact`

The translation function starts code generation for tag tests with emission of the byte code instruction `testtag t table`. The arguments specify the register that contains the test value and the address of the hash table in the immediate environment. The code for the `else`-branch is emitted directly after the test instruction. This saves an extra argument for the location of the `else`-branch. Finally, the code for all other branches is created with the procedure `transl_branch` that is applied to every entry in the branch vector. The tag of each entry is used as hash key under which the start address of the branch code is stored. The binding is generated at the beginning of the branch code and the translation function is recursively called to generate code for the successor instructions.

```
| transl TestTag(testval,branches,elseInstr) =
  let val _      = Registers.reset()
      val t      = loadIdRef testval
      val map    = Map.map ()
      val table = ienv_insert map

      fun bind (_,Wildcard) = ()
        | bind (i,IdDef id) =
          let val r = Registers.fromId id
          in  EMIT (load_tagval r t i)
          end

      fun transl_branch (tag,defs,next) =
        (Map.insert(map,tag,PC);
         Vector.appi bind defs;
         transl next)
  in
    EMIT (testtag t table);
    transl elseInstr;
    Vector.app transl_branch branches;
  end
```

For exhaustive tests, the jump table is embedded into the code. The table is allocated directly behind the test instruction and stores the jump offsets to the code of the branches.

```
| transl TestTagCompact(testval,branches) =
  let val _ = Registers.reset()
      ... (* load testval and define 'bind' *)
      val startPC = ref 0
      fun transl_branch (defs,next) =
         let val offset = PC - !startPC
         in  ... ;  (*emit offset into table*)
             Vector.appi bind defs;
             transl next
      end
  in
    EMIT (ctesttag t);
    startPC := PC;
    (*allocate space for inlined table*)
    PC := PC + length branches;
    Vector.app transl_branch branches
  end
```

Procedure Application: `Apply`, `ApplyTail`, `Return`

To call a procedure, the closure and all arguments have to be loaded into registers. Everything else is done by the call instruction: it sets the arguments to the global registers bank and delegates control to the scheduler. When the procedure returns, the scheduler registers might contain some results. The procedure `transl_ccc` generates code that transforms these values into the expected format and stores them into byte code registers.

```
| transl Apply(closure,args,(defs,next)) =
  let val _  = Registers.reset()
      val r0 = loadIdRef closure
      val #[r1,...,rn] = Vector.map loadIdRef args
  in
    EMIT (seam_call r0 r1 ... rn);
    transl_ccc defs;
    transl next
  end
```

Tail calls are even simpler to translate because they do not have a continuation. The translation function only needs to load the closure and the arguments into registers and emit the `seam_tailcall` instruction. The translation of the return instruction works in the same way. All arguments are loaded into registers and the byte code instruction is emitted.

```
| transl (Return args) =
  let
    val _  = Registers.reset()
    val #[r1,...,rn] = Vector.app loadIdRef args
  in
    EMIT (seam_return r1 ... rn);
  end
```

Exceptions: `Raise`, `Try`, `EndTry`

There is a direct correspondence between the `Raise` instruction of the abstract code and the `raise` instruction of the byte code. This is an easy job for the compiler.

```
| transl (Raise e) =
  let
    val _ = Registers.reset()
    val r = loadIdRef e
  in
    EMIT (raise r);
  end
```

The code block in which a handled exception might occur is surrounded by `Try` and `EndTry`. The first one pushes the exception handler onto the handler stack and the second one pops the handler if no exception occurred. The translation function creates the handler procedure, which consists of a binder for the exception value and the start address of the handler code.

```
| transl Try(body,(def,handlerInstr)) =
  let
    val _       = Registers.reset()
    val handler = ienv_insert (0,0) (*insert dummy*)
  in
    EMIT (install_handler handler);
    transl body;
    (*patch handler after the start address is known*)
    ienv_update (handler, (getIdDef def,PC));
    transl handlerInstr;
  end
| transl EndTry    = EMIT remove_handler
```

The body code precedes the handler code. When the compiler emits code to install the handler, it does not know its start address. Therefore, it patches the handler when the information is available.

Join Points: `Shared`

The `Shared` nodes explicitly represent the join points in the abstract code graph. Of course, the compiler must not generate several copies of the code below a join point. Hence, it memorizes the start address s of the byte code when it compiles the abstract code below a join point. When it visits the join point for a second time, it emits a jump to s.

```
| transl Shared(stamp,next) =
  (Registers.reset();
   if visited stamp then
     let
       val offset = get_offset stamp
     in
       EMIT (jump offset)
     end
   else
     (set_visited stamp; transl next)
  )
```

5.4 Compiler

This section defines the compiler structure that offers exactly one procedure to the outside world. `compile` receives a procedure code, as it is defined in Figure 3.1 (page 15). The abstract code is compiled by using all helpers specified in the previous sections.

```
structure Compiler :
  sig
    val compile : procedure -> byte_code * immediate_env * int
  end
  =
  struct
    ... (*all definitions of the previous sections*)
    fun compile Procedure(body,formalArgs,numberOfIds,_,_) =
        let
          structure Registers = Registers(val min = numberOfIds
                                          fun mapping id = id)
        in
          codebuffer_init();
          offsettable_init();
          ienv_init();
          transl_ccc formalArgs;
          transl body;
          (code,ienv,Registers.peak())
        end
   end
```

Figure 5.1: byte code compiler

Figure 5.1 shows the definition of the byte code compiler. First, the compiler creates a structure for register handling. The simple version uses the identity function to map local identifiers to byte code registers. In the second step, all data structures are initialized and the calling convention is compiled for the formal arguments of the procedure. After these preparation steps, the compiler translates the abstract code graph into byte code that is stored in a code buffer. Finally, the compiler returns the byte code together with the immediate environment and the number of byte code registers that the procedure needs at run-time.

Notice that this compilation scheme also works for a static compiler. So up to now, the compiler does not use run-time information. However, the optimizations covered in the next chapter capitalize on run-time information about immediate values.

Task Creator

When the scheduler assigns a closure to the byte code task creator, the task creator invokes the compiler to transform the abstract code into byte code. The compiler returns a reference to the compiled byte code (BP), a reference to the immediate environment (IP), and the maximal number of needed registers (*peak*). The task creator computes the size of the task, which is $peak + 6$. The responsible interpreter is the byte code interpreter and the closure pointer (CP) is also known since it is passed as an argument to the task creator.

Triggering Compilation

In order to trigger compilation, the run-time system has to detect when a procedure is executed for the first time. An early version of the byte code compiler adopted the technique of the native code compiler. When the procedure is loaded, the code constructor creates a by-need that contains the abstract code and all information that the compiler needs to translate the code into byte code. Before the task creator is invoked, the scheduler requests the code and forces evaluation, which means that it triggers compilation to byte code. This triggering policy is nice because it makes use of the existing architecture. However, the final version of the byte code system implements a different approach that is more flexible. This so called selective compilation is presented in the following chapter.

6 Compiler Optimizations

The compilation scheme presented in Chapter 5 offers much room for optimization. This chapter deals with optimizations that the byte code compiler applies to ensure space efficiency and fast execution of the generated code. A set of small optimizations is explained at the beginning. Thereafter, *selective compilation* is introduced, which is a flexible approach to decide when to compile a procedure. Furthermore, the compiler utilizes advanced techniques such as register allocation, procedure integration, also known as inlining, and self call optimization.

6.1 Small Optimizations

An optimization is called *small* if it produces low compilation overhead and is easy to implement. This section presents small optimizations for the treatment of scratch registers. Additionally, the compiler can do more clever with tag tests, and a peephole optimizer locally improves code quality. Furthermore, the compiler tries to eliminate calling convention tests.

Scratch Registers

In the previous chapter, the translation function frees all scratch registers when it compiles a new abstract code instruction, but it does not free scratch registers in between. However, there is, for instance, only one scratch register needed to initialize a tuple of arbitrary length. Therefore, we extend the `Registers` functor with the following procedure:

```
val free_scratch : register -> unit
```

The call `free_scratch R4` declares the register `R4` to be free, which means that it can be reused when `new` is called the next time. The procedure only affects scratch registers and nothing happens when a normal register is passed. We can now refine all initializations in the translation function as exemplified with `NewTup`.

```
fun transl NewTup(dst,fields,next) =
  let
    val t     = getIdDef dst
    val arity = Vector.length fields

    fun init (i,x) =
        let
          val r = loadIdRef x
        in
          EMIT (init_tup t r i);
          Registers.free_scratch r (*this is new*)
        end
  in
    EMIT (new_tup t arity);  (*creation*)
    Vector.appi init fields; (*initialization*)
    transl next
  end
```

Tag-Tests

The abstract code offers two ways to test a tagged value: general tag tests and compact tag tests. The byte code offers two corresponding instructions and additionally provides a special instruction for exhaustive tests, which are compact tests without a range check. For performance reasons, it is desirable to compile as many tests as possible into exhaustive tests. Profiling revealed that there are often only three or four cases missing in order to make the test exhaustive. The byte code compiler detects such situations, extends the test table and fills the new entries with the `else`-branch offset as jump target.

The native code compiler does not use inlined test tables. Instead, it takes Alice vectors and thus, because of the immediate environment, introduces an additional indirection.

The static Alice compiler breaks down complex pattern matching into several tests. This explains why unary tag tests are frequent. A specialized instruction handles them efficiently.

Peephole Optimization

Peephole optimization is a classical technique to locally improve code quality. The compiler tries to detect small sequences of instructions that cancel each other or that can be replaced with one specialized instruction. This kind of optimization is especially effective if the instruction set consists of many fine-grained instructions. Nevertheless, generated byte code also contains some inefficiencies that can easily be fixed. For instance, the following sequence occurs frequently

```
load_int R0 0
await R0 (*can be removed*)
```

Obviously, the await instruction is superfluous, as the number 0 is not a transient. If the liveness information of the abstract code indicates that `R0` is never used in the program again, the `load_int` instruction can also be removed.

Another sequence for which the compiler generates a specialized instruction is

```
load_int R0 0
seam_return R0
```

These instructions are replaced by `seam_return_unit`.

Omission of Calling Convention Tests

The instructions `ccc1` and `cccn` combine parameter passing and calling convention conversion. An optimizing compiler can remove the conversion tests when it is sure that the number of supplied arguments always coincides with the expected number of arguments. This is difficult to find out at the entry point of a procedure. One can only remove the test if all call-sites are known at compile-time. However, if a callee returns to its caller, the analysis is straightforward. Suppose the following byte code implements the callee:

```
(*procedure f*)
ccc1 R0
...
return R0
```

The call site, generated by the simple compiler, looks as follows

```
...
bci_call f R0 R1 (*byte code call*)
ccc1 R0          (*conversion test*)
...
```

The optimizing compiler uses the `outArity` information (Figure 3.1, p. 15) of procedure `f` to generate a specialized instruction `getarg1` instead of `ccc1`. For `cccn`, there is also a specialized version without test. This simple strategy saves nearly half of all conversion tests (47% for the Alice Toplevel interpreter).

6.2 Selective Compilation

Up to now, a procedure is compiled when it is executed for the first time. This section introduces a more general model in which compilation is expressed as transition between two forms of code, controlled by arbitrary boolean conditions. This gives fine grained control over compilation and enables the compiler to apply expensive optimizations selectively to procedures that are often executed.

In Chapter 5, the compilation strategy was based on by-need evaluation of code. A by-need can be seen as a value that changes its appearance at most ones. It starts to be a placeholder associated with a first class computation. As soon as a thread requires the actual value of the placeholder, the scheduler triggers computation and fills the placeholder with the result. Figure 6.1 depicts the evolution of by-need code into concrete code for procedure p as a transition system. There is something special about

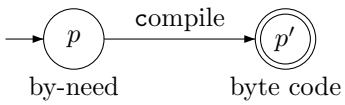


Figure 6.1: evolution from a by-need to actual code for procedure p

by-need code: it stores the abstract code that is already an executable code format. In order to reduce compilation overhead, the abstract code interpreter can execute the procedure for the first n times, and compilation is triggered when the code is requested for the $(n+1)$-th time. More generally, an arbitrary number of transformation steps can be expressed. For instance, the compiler can apply sophisticated optimizations if a threshold $m > n$ is exceeded. Figure 6.2 depicts the resulting transition system. c counts the number of calls to procedure p.

Unfortunately this is not expressible with transients. The virtual machine does not provide services for conditional transitions. We have to implement triggering of transitions manually.

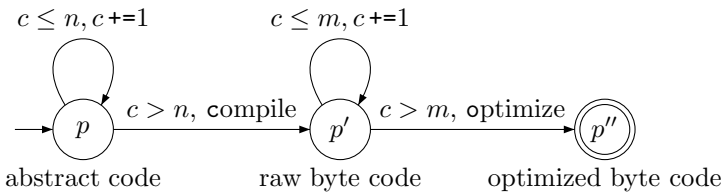


Figure 6.2: deferred compilation of code

Implementing Selective Compilation

The control logic of the transition system is implemented in special task creators. One task creator is defined for every state in the transition system. The new task creators are implemented as wrappers around the original ones. They check the transition condition, and if it is met, switch to the successor state. Finally, they call the original task creator to push the new task. As no information is collected as soon as the final state is reached, the wrapper can be removed and the original task creator is used.

Selective compilation makes optimization of intra-language invocations more difficult. As compilation can be arbitrarily postponed, the compiler may only find out that a callee may eventually have a byte code representation. In this case, the compiler generates a rewrite call that specializes itself as soon as the callee is compiled.

6.3 Capitalizing on Immediate Values

Immediate values are a potential source for optimizations. If primitives and procedures are immediate, i.e. known at compile-time, the compiler can generate more efficient code by using specialized instructions. Due to dynamic linking, the static compiler cannot collect much information about values. At run-time, the dynamic compiler can access much more immediate values. The Alice system uses template-based compilation to further increase the number of available values.

6.3.1 Template-Based Compilation

The definition of procedure code (Figure 3.1, p. 15) contains a partial mapping *subst* from global variables to values. We extend the compiler to make use of this substitution by considering the procedure code as a template. The global variables are template variables that the substitution instantiates during compilation.

Finding immediate values is especially interesting for the translation of the `Apply` instruction. Whenever the translation function detects that the closure argument is immediate, which means that it does not change at run-time, it can create more efficient code. The extended translation function is depicted in Figure 6.3.

It is out of the scope of this thesis to completely specify `translImApply`. We only give an informal overview about the different cases. For all immediate procedures, the compiler extracts the concrete code out of the closure and reflects on the type of this code. If it is byte code, the byte code call instruction can be used. In case it is abstract code, the compiler emits a rewrite call. The call might be recursive. More details on optimizations for recursive calls are given in Section 6.6. The compiler can

```
| transl Apply(closure,args,(defs,next)) =
  let
    fun extractIm (Immediate v) = SOME v
      | extractIm (Global id)   = subst id
      | extractIm (Local _)     = NONE

    val #[r1,...,rn] = Vector.map loadIdRef args
  in
    case extractIm closure of
        NONE   =>
          let val r0 = loadIdRef closure
          in  EMIT (seam_call r0 r1 ... rn)
          end
      | SOME v => translImApply (v,r1,...,rn)
    transl_ccc defs;
    transl next
  end
```

Figure 6.3: optimizing compilation of `Apply`

reflect on primitives, for instance to generate `iadd` instead of a call to `op+`. All other calls to immediate procedures are translated as immediate calls.

6.3.2 Closure Specialization

The substitution is constructed and modified when a new closure is created. The idea of closures is to parametrize code over its environment, such that different closures can share the same code. In the following example, each invocation of `add` creates a new closure with different definitions of `x`, but with the same code `x+y`. In particular, the just-in-time compiler translates the code at most once, which is especially important if `add` is called frequently.

```
val a = 1
...
val add x = fn y => x+y
val inc = add a
```

However, for the following equivalent definition of `inc`, it is clear that the body `x+a` can only belong to a single closure.

```
val a = 1
...
val inc = fn x => x+a
```

Therefore, it is save to substitute the global variable `a` by 1, which means that the closure is specialized to a particular environment.

The abstract code offers, besides the usual way to create closures with `NewClosure`, an instruction `NewSpec` that creates a closure specialized with respect to a specific environment.

instr ::= ...
| `NewSpec` {`dest` : *def* `globals` : *ref** `code` : *code* `next` : *instr*}

On execution, `NewSpec` writes all global values into the substitution of the procedure in `dest`, such that the compiler can replace all global variables by immediate values.

Since the code is specialized to the current environment, the code cannot be shared, but has to be compiled separately for every closure.

The static Alice compiler decides upon the placement of `NewSpec` and `NewClosure` with a simple heuristic. For each closure that is created on top-level – so each closure that is created only once when the component is loaded – it uses `NewSpec`. In all other cases, it generates `NewClosure`.

6.4 Register Allocation

At a first glance, it seems unnecessary to consider register allocation for the byte code. The machine model offers an unbounded number of registers and the compiler can therefore map every local variable of the abstract code to a byte code register. However, programs in abstract code are represented in static single assignment form (Section 3.1, p. 16). SSA form highly increases the number of local variables, which means that the compiler produces big tasks if it uses a one-to-one mapping. Pushing a big task is more expensive than pushing a small one, and a big stack slows down garbage collection since all task stacks reside in the abstract store. Register allocation reduces task sizes and therefore speeds up execution.

Liveness Intervals

The static Alice compiler computes a liveness interval for each local variable. Each abstract code instruction is assigned to natural number, its program point. The numbering defines a topological order on the abstract code graph. An interval is a pair of program points. For $x \mapsto [a, b]$, a indicates the program point where x is defined and b gives the program point where x is used for the last time. In the context of an unbounded number of byte code registers, allocation means to find the minimum number of sets with non-overlapping intervals.

Linear Scan Register Allocation

The classical register allocators of static compilers reduce register allocation to a graph coloring problem which can be approximated efficiently (Chaitin [15]). For just-in-time compilers, the algorithm is too slow because it works on a register interference graph that can be quadratic in the number of liveness intervals. Poletto and Sarkar invented a "linear scan register allocation" [34] that is linear in the number of liveness intervals and does not require expensive graph construction. The assumption of a fixed number of registers is essential for the linear running time. Let V be the number of liveness intervals. For an unbounded number, the algorithm can be implemented with a worst case time complexity of $O(V \cdot \log V)$.

Figure 6.4 shows the main procedure of the allocator in ML-like notation. The allocator starts with a canonical one-to-one register mapping. The number of needed registers is set to zero and a new min-heap (cf. Cormen et al. [16]) is created and initialized. The heap contains registers together with the end point of their interval. The minimum of the heap is the register with the smallest end point. So the register that is most likely to expire is always on the top of the heap. The allocator loops over all intervals in order of increasing start point. It first takes the minimum from the heap and checks if its end point overlaps with the start point of the current interval. If not, the minimum is deleted and the register is reused for the current variable. Otherwise, the variable is assigned to a fresh register and the number of needed registers is incremented. In

```
fun allocator(intervals,numberOfIds) =
  let
    val mapping = Array.tabulate (numberOfIds, fn x=>x)
    val heap    = Heap.heap()

    fun alloc((id,a,b),max) =
        let
          val (b',reg) = Heap.min heap
        in
          if b' < a then  (*register can be reused*)
            (Heap.deleteMin heap;
             Array.update(mapping,id,reg);
             Heap.insert(heap,(b,reg));
             max)
          else            (*a fresh register is needed*)
            (Array.update(mapping,id,max);
             Heap.insert(heap,(b,max));
             max+1)
        end

    val max_regs = foldl alloc 0 intervals
  in
    (mapping,max_regs)
  end
```

Figure 6.4: register allocator; intervals are sorted in order of increasing start point

both cases the newly occupied register is added to the heap together with its end point, and the mapping is updated accordingly. After the loop, the final mapping and the maximal number of registers, i.e. the space needed in the stack frame for local variables, is returned.

Analysis

Computing and sorting of liveness intervals happens statically. The allocator loop (hidden in `foldl`) takes linear time in the number of liveness intervals. Extracting the minimum from a min-heap is a constant time operation. In worst case, if all intervals overlap, the heap contains all intervals. A heap is a complete binary tree and insertion and deletion maximally have to walk along the longest path in the tree. Therefore, they take $O(\log V)$ time. In summary, this leads to $O(V \cdot \log V)$ time complexity.

A binary heap can be implemented as an array. Computing the indexes of parents or children is realized by efficient bit shift operation. In practice, the number of intervals is smaller than 20 for most procedures. We discovered only a few procedures with about 1000 intervals. So the algorithm nearly performs linear time for standard Alice programs. Table 6.1 shows the register need of the Alice Toplevel interpreter. Each procedure was fully byte compiled on the first execution.

	Toplevel interpreter
without allocation	96 381
with allocation	24 729
space savings	74%

Table 6.1: number of local variables with and without register allocation

Register allocation in the native code compiler takes $O(V^2)$. The reason is that it

performs two allocations in parallel. First, hardware registers are allocated and spilled if necessary. This is linear time as only three hardware registers are used. Second, the allocator naively checks if a spilled register is expired and its memory slot can be reused. The check is implemented as linear search in a singly linked list.

Compiler Extension

The register allocator can easily be incorporated into the compiler. The allocator only needs the liveness intervals that are stored in every procedure code. It returns a mapping, represented as an array, from local identifiers to byte code registers and the number of local registers. This information is passed to the registers functor.

```
fun compile Procedure(body,formalArgs,numberOfIds,
                      subst,liveness) =
  let
    val (map, max_regs) = allocator(liveness,numberOfIds)
    structure Registers = Registers(val min = max_regs
                                    fun mapping id = sub (map,id))
    ...
```

6.5 Procedure Integration

Abstraction is a key feature of modern programming languages. In functional languages, procedures are the basic abstraction units and higher-level mechanisms, like modules, can be compiled into procedures. This is exactly what the static Alice compiler does when it transforms the source language into abstract code.

Procedure calls cause overhead: a new task has to be created for each call, and the parameters are passed from the current task to the new one. In a system where procedures are ubiquitous, it is essential to reduce the overhead as much as possible. This section discusses *procedure integration* for Alice Abstract Code. Procedure integration is a powerful optimization for programs that use procedural abstraction. The first part of this section focuses on the question which procedure to integrated, whereas the second part shows how to integrate.

In the literature, the term *inlining* is more common. We prefer the notion of *integration* because the term *inlining* is highly overloaded (inlining of procedures, primitives, byte code, ...).

Motivating Example

The idea behind procedure integration is quite simple. Suppose we have the following two procedures:

```
fun do_magic x = x+1

fun f nil = nil
  | f (x::xr) = do_magic x :: f xr
```

If we integrate `do_magic` into `f`, we arrive at the following implementation for `f`:

```
fun f nil = nil
  | f (x::xr) = x+1 :: f xr
```

Experiments show that the alternative implementation is about 60% faster. However, the resulting code sacrifices abstraction for efficiency. It is a much better idea to keep the abstraction in the source language, but automatically remove it in the internal code. This is what procedure integration is all about.

General Picture

The analysis phase decides which procedure should be integrated. Integration means to substitute the call instruction by the callee's body. Additionally, parameter passing is simulated. So procedure integration goes in four phases:

1. decide which procedure to integrate
2. compile parameter passing from caller to callee
3. compile callee's body
4. compile parameter passing from callee to caller

Fortunately, there is no difference between parameter passing from caller to callee, which is needed for procedure calls, and parameter passing from callee to caller, which is needed for procedure returns. The same algorithm can be used for both scenarios.

6.5.1 Which Procedure to Integrate?

A callee is only integrated into the caller if it does not exceed a size limit. Since an integrated callee is specialized to a specific call site, every instance of the procedure has to be compiled separately. Compilation time T_C is strictly increasing in the size of a procedure. Suppose procedure p needs time $T_C(p) = |p| \cdot t$ to be compiled, where t is some time unit, for example $1\,ms$. If procedure q contains n calls to procedure p that are all integrated, it takes $(|q| + n \cdot |p|) \cdot t$ to compile q. The size limit prevents arbitrary growth of code size that would significantly slow down compilation.

Additionally, integration is only enabled for the transition to the final state of the code transition system. This way integration is only performed for procedures that are often executed. It remains to show how the size of a procedure is defined and how the information is collected.

The Size of a Procedure

The analysis phase works on the abstract code. There is a close correspondence between byte code and abstract code. The size of a procedure is defined as the number of nodes in the abstract code graph. There are a few exceptions to this simple rule. Some node types do not produce byte code and are therefore not counted. Other types of nodes produce lots of instructions and therefore weigh more than standard nodes. The exact definition of the size is an experimental task.

Collecting Information

The analysis phase computes for every procedure a set of pointers to integration candidates. This works by stepwise inspection of the global call graph. An example is given in Figure 6.5. If the analysis starts at g, it transitively analyzes procedures h, i, j, and k and stores the information at each visited node. When f is analyzed afterwards, the information for h is already available. This ensures that every edge in the global call graph is visited at most once.

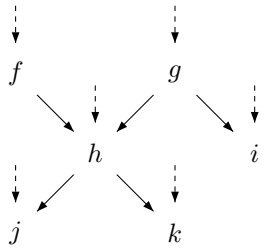


Figure 6.5: call graphs for a set of procedures

Call graphs are inherently cyclic. We have to take care that the analysis phase does not diverge. Whenever the analyzer revisits a node, it stops to break the cycle. Figure 6.6 shows two types of cyclic dependencies between procedures. If the analysis starts at f, it does not follow the dashed edges. Interestingly, the mechanism transforms

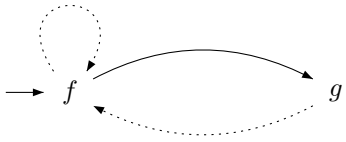


Figure 6.6: cyclic dependency between procedure f and g

mutual recursion into self recursion if the procedures do not exceed the size limit. The compiler can generate more efficient code for self recursion, as we will see in Section 6.6. A predicate that checks if an integer number is even can be implemented with two mutually recursive procedure in the following way

```
fun even 0 = true
  | even n = odd (n-1)
and odd 0 = false
  | odd n = even (n-1)
```

The byte code compiler removes the mutual recursion and generates byte code that closely corresponds to the following source code

```
fun even 0 = true
  | even n =
    let val n' = n-1 in
       case n' of
         0 => false
       | 1 => even (n'-1)
fun odd 0 = false
  | odd n = even (n-1)
```

Notice that `odd` is not transformed. The analysis starts at `even` and transitively analyzes `odd` where it stops to break the cycle. There is no extra analysis for `odd`. However, the only thing one can save by an analysis of `odd` is a single call to `even`.

Of course, one could define more relaxed termination conditions. An interesting extension to the currently used condition would be to unroll self loops until the size limit is reached.

Closure Creation Considered Harmful

Integration of procedures that create closures can be harmful. If a procedure integrates its own creation, the code is newly compiled on every call of the procedure. This can even cause the compiler to diverge if compilation is not delayed like in Alice.

Let us consider a simple example to illustrate the problem.

```
fun f x =
  let fun g y = if x=0 then y else (f (x-1)) y
  in  g
  end
```

Procedure `f` creates a closure for its inner procedure `g` and returns it. When procedure `g` is compiled, the integration analysis detects that `f` can be integrated since it is small enough and because there is no cycle in the call graph. Inside `g` there is a new closure created (application `f (x-1)`) that uses the same code as `g`. So we integrated closure creation into the loop. This produces enormous compilation overhead for curried procedures. To avoid this effect, procedures that create closure are not considered as integration candidates.

A more general solution would be to check for each candidate that contains closure creation whether the closure contains the same code as a procedure in the currently investigated call graph. The analysis allows to integrate the first call to `f`, but all recursive call are not integrated. We suppose that the savings of a single call cannot amortize the additional analysis overhead.

Related Work

The Jalapenño Java virtual machine [7] searches for hot call edges, i.e. caller-callee pairs, to guide integration decision. In essence, this means that a call graph is periodically constructed at run-time. Hölzle and Ungar proposed a more lightweight approach for the programming language *Self* [23]. The authors use the procedure call stack as basis for their inline decision and thus do not need to construct an extra graph. The stack corresponds to a path in the call graph. So the analysis is split into several inspections of the call stack. Both approaches consider edge weights of the call graph to find out good inline candidates.

In contrast to that, the analysis of the Alice byte code system is more coarse-grained and does not consider edge weights. The design focus clearly lies on the speed of the analysis and not on the exactness. It would be interesting to experiment with more exact techniques.

6.5.2 In-Place Parameter Passing

Parameter passing has to respect the calling convention of Alice. The convention prescribes that the receiver converts the arguments into the format it expects. If we integrate a callee into a caller, we do not need any checks at all. Both parties know exactly in which format they require the arguments and the calling convention conversion can be hardwired into the code.

If the number of formal arguments is not equal to the number of actual arguments, compilation of the calling convention is straightforward. We can simply construct or deconstruct a tuple into the argument registers. Let us take an example procedure `f` that is called by another procedure `g`. `f` expects a tuple as argument, but `g` supplies

two discrete arguments. `ccc1` automatically constructs a pair.

```
code for f:                                  code for g:
  ccc1 R0                                      ccc1 R0
  ...                                          load_global R1 0 (*load f*)
  select_tup0 R1 R0 (*R1<-#1 R1*)              seam_tailcall R1 R0 R0
  iinc R1
  seam_return R1
```

When `f` is integrated into `g`, the pair is constructed explicitly.

```
code for g' with f integrated:
  ccc1 R0
  new_pair_init R1 R0 R0  (* R1 <- (R0,R0) *)
  ...
  select_tup0 R2 R1  (* R2 <- #1 R1 *)
  iinc R2
  seam_return R2
```

If number of actual arguments coincides with the number of formal arguments, things get more complex.

```
code for f:                                  code for g:
  cccn R0 R1                                   cccn R0 R1
  ...                                          load_global R2 0
  return R2                                    seam_tailcall R0 R1 R0
```

The most obvious solution is to use register move instructions.

```
code for g' with f integrated:
  cccn R0 R1
  load_reg R2 R1 (* R2 <- R1 *)
  load_reg R3 R0 (* R3 <- R0 *)
  ...
  seam_return R4
```

This causes some problems because, in the original version, the call instruction sets all arguments at once. Thus, parameter passing is atomic. Suppose there is some clever register allocation involved. The allocator will detect that `R0` and `R1` stop to live at the beginning of the body of `f` and we get the following code:

```
code for g' with f integrated + register allocation:
  cccn R0 R1
  load_reg R0 R1 (* R0 <- R1 *)
  load_reg R1 R0 (* R1 <- R0 *)
  ...
  seam_return R2
```

Clearly, this is wrong because the value of `R0` is written before it is read in the assignment to `R1`. We could adjust the liveness information to avoid such conflicts. However, this increases the number of local registers and impedes further optimizations (e.g. elimination of load instructions, Section 6.5.3). We can do more clever. The algorithm has to detect *write-read conflicts* and resolve them in some way. For the above example we can write the following code to resolve the conflict:

```
code for g' with f integrated + register allocation:
  cccn R0 R1
  load_reg S0 R0 (* S0 <- R0 *)
  load_reg R0 R1 (* R0 <- R1 *)
  load_reg R1 S0 (* R1 <- S0 *)
  ...
  seam_return R2
```

In general, there are two types of conflicts that arise in parameter passing:

1. wrongly ordered assignments

$$\begin{array}{lcl} R_0 & \leftarrow & \top \\ R_1 & \leftarrow & R_0 \\ & \cdots & \\ R_n & \leftarrow & R_{n-1} \end{array}$$

2. cyclic assignments $R_0 \leftarrow \cdots \leftarrow R_n \leftarrow R_0$

The algorithm to compile in-place parameter passing has to identify these conflicts, resolve them, and finally generate code that simulates parameter passing. The following section develops an efficient solution to that problem.

Write-Read Conflict Resolution

The algorithm works in two phases. First it decomposes the assignments into a list of marked assignment chains. The example set of five assignments

$$\begin{array}{lcll} 1: & R_0 & \leftarrow & 1 \\ 2: & R_1 & \leftarrow & R_3 \\ 3: & R_2 & \leftarrow & R_0 \\ 4: & R_3 & \leftarrow & R_4 \\ 5: & R_4 & \leftarrow & R_1 \end{array}$$

makes up two chains:

$$\begin{array}{lll} 1': & R_2 \leftarrow R_0 \leftarrow 1 & \text{(normal)} \\ 2': & R_1 \leftarrow R_3 \leftarrow R_4 \leftarrow R_1 & \text{(cyclic)} \end{array}$$

Definition-use chains are used to construct assignment chains. The algorithm takes an assignment and transitively follows each use of the destination register as well as the definition of the source register. If the algorithm encounters that the start of the chain equals its end, it stops and marks the chain as cyclic. In total this needs linear time in the number of assignments.

In the second phase the algorithm generates code for the assignment chains. For normal chains, this works by a forward scan through the chain. For $1'$ we get:

```
load_reg R2 R0 (* R2 <- R0 *)
load_int R0 1  (* R0 <- 1  *)
```

For cyclic chains we also do a forward scan, but we split the first assignment into two by introducing a scratch register. So we get the following code for $2'$:

```
load_reg S0 R3 (* S0 <- R3 *)
load_reg R3 R4 (* R3 <- R4 *)
load_reg R4 R1 (* R4 <- R1 *)
load_reg R1 S0 (* R1 <- S0 *)
```

In the actual implementation cyclic chains are resolved with swap instructions, as this requires less byte code instructions.

6.5.3 Eliminating Load Instructions

Naive compilation of parameter passing for integrated procedures will not result in good code. The increment procedure in Figure 6.7 serves as a running example. The dereferencing operator that Alice implements as a procedure can be integrated. Naive

```
fun !(ref x) = x (*deref defined in Alice library*)
  ...
fun inc r = r:=!r+1
```

Figure 6.7: increment of an integer reference cell

simulation of parameter passing for the `!` operator results in the following code:

```
ccc1 R0
load_reg R1 R0   (* incoming arguments *)
load_cell R2 R1  (* body of !          *)
load_reg R3 R2   (* outgoing arguments *)
iinc R3
set_cell R0 R3
seam_return_unit
```

Two thirds of the integrated code constitute of superfluous parameter passing. What we actually want is code like this:

```
ccc1 R0
load_cell R1 R0  (* body of ! *)
iinc R1
set_cell R0 R1
seam_return_unit
```

In general, all superfluous load instructions should be eliminated. Two techniques are used to tackle this issue. The register allocator can eliminate many load instructions. If an actual argument stops to live when it is passed, register allocation can use this register for the formal parameter and this way no register move is needed. If the

liveness of an actual arguments overlaps the whole body of the callee, a restricted form of alias analysis is required.

Merging Liveness Intervals

The register allocator can eliminate load instructions automatically if it uses a modified liveness information, i.e. the liveness intervals of a procedure plus all its integrated callees. To get this extended liveness information, one could do a complete liveness analysis for each procedure after integration; but experiments show that this considerably slows down execution. It is cheaper to merge the liveness intervals that the static compiler provides.

To illustrate how the algorithm works, we consider the increment example again. Every instruction is annotated with its program points.

```
code for inc without integration of !:
 0:    ccc1 R0
 1:    immediate_call ! R0
 2:    ccc1 R1
 3:    iinc R1
 4, 5: set_cell R0 R1
 6:    seam_return_unit

code for !
 0:    ccc1 R0
 1, 2: load_cell R1 R0
 3:    seam_return R1
```

The procedure integration phase merges sets of liveness intervals. For the example, the first set is $F \stackrel{\text{def}}{=} \{R_0 \mapsto [0,5], R_1 \mapsto [2,4]\}$ and the second set is $S \stackrel{\text{def}}{=} \{R_0 \mapsto [0,1], R_1 \mapsto [2,3]\}$. To avoid name clashes, the registers in S are α-renamed to R_2 and R_3. After integration, the code and the corresponding program points look as follows:

```
0:    ccc1 R0
1, 2: load_reg R2 R0   (* incoming arguments *)
3, 4: load_cell R3 R2  (* body of !          *)
5, 6: load_reg R1 R3   (* outgoing arguments *)
7:    iinc R1
8, 9: set_cell R0 R1
10:   seam_return_unit
```

We are looking for an algorithm that merges F and S. The liveness intervals of the resulting set T should be consistent with the program points in the example above. Of course, the annotated program points are not known statically. They are only given to check the output of the algorithm.

In order to get the liveness intervals T for registers R_0, R_1, R_2, and R_3, the algorithm performs the following steps:

1. Examine the original caller to find the program point p_{call} of the call exit (`ccc1` instruction after the call instruction).
   example: $p_{call} = 2$
2. Find out the number of program points p_{max} of the callee.
   example: $p_{max} = 4$
3. Add p_{call} to interval bounds in S.
   example: $S' \stackrel{\text{def}}{=} \{R_2 \mapsto [2,3], R_3 \mapsto [4,5]\}$.
4. Add p_{max} to all interval bounds in F that are greater than p_{call}.
   example: $F' \stackrel{\text{def}}{=} \{R_0 \mapsto [0,9], R_1 \mapsto [6,8]\}$
5. $T \stackrel{\text{def}}{=} F' \cup S'$
   example: $T = \{R_0 \mapsto [0,9], R_1 \mapsto [6,8], R_3 \mapsto [2,3], R_4 \mapsto [4,5]\}$

All steps of the algorithm, except the first one, only work on the liveness intervals. To find p_{call}, the program point for the call instruction has to be computed. Fortunately, an extra pass through the caller code is not required. The program points are counted during the search for integration candidates.

Some effort is necessary to gain an implementation that is efficient, when several procedures are integrated. Step four of the algorithm is the main source of improvement. A better strategy is to collect the offsets p_{max} for every p_{call} and shift all intervals of T at the end. Then this algorithm is linear in the maximum of the number of program points (of the caller after integration) and the number of liveness intervals (of the caller and all integrated callees).

In general, this approach works well for all load instructions at the exit of integrated procedures, but it cannot eliminate the first load instruction in the increment procedure.

Alias Analysis

The first load instruction is generated because the liveness intervals of R_0 and R_3 overlap. However, it is obvious from the example that this instruction is superfluous. It just creates a copy that the callee can only read. We need a technique to detect so called *aliases* for registers whose liveness intervals overlap. For incoming arguments the algorithm can be integrated into the merging phase of liveness intervals.

1. For every incoming argument R_i, check if its liveness interval overlaps the starting point of the formal parameter S_j. If this is the case, do not add a liveness interval for R_i, but remember the mapping $S_j \mapsto R_i$.
2. Combine the alias mapping with the mapping that the register allocator returns.

This way we finally get rid of the first load instruction.

There is an interesting extension to the alias analysis. Suppose an alias is passed to an integrated procedure and this procedure returns the alias again. Then the information about the alias is lost at the exit of the integrated callee and the caller requires a superfluous load instruction. Our algorithm can be extended in the following way:

- Introduce a local alias mapping for every procedure exit.
- On every exit check for every return argument if it is an alias. If this is the case, store this information in the local alias mapping.

- When all exits were visited, compute the greatest lower bound over all local alias mappings.
- Combine the result with the global alias mapping, i.e. the mapping that is used for aliases of incoming arguments.

6.5.4 Compilation of the Body

The previous sections explained the analysis phase and showed how parameter passing can be implemented efficiently. This section deals with code generation of the body of an integrated procedure.

The analysis phase passes the collected information to the compiler. Whenever the compiler detects a procedure application, it looks up if the procedure is an integration candidate. If it is one, parameter passing is compiled and the procedure code of the candidate is fed to the compiler. The procedure code still contains the original identifier names. The compiler takes care that the local identifiers are synchronized with the names computed by the merging algorithm for liveness intervals.

Since the context of the integrated callee is fixed, global variables are converted into immediate values. So the compiler specializes the integrated procedure to its caller. This not only elegantly treats global variables, but more importantly makes more immediate values accessible to the compiler. As outlined in Section 6.3, the compiler can, for instance, generate more efficient code for calls to immediate procedures [1].

All instructions that exit an integrated callee have to be converted into "intra-caller" jumps. For instance, the meaning of a return instruction is that the callee jumps to the continuation of the caller. Obviously, if the callee is integrated into its caller, it is wrong to compile the return instruction literally since it would also exit the caller. A return instruction is therefore translated into a jump to the continuation of the caller. Another kind of exit is a tail call, which is converted into a normal call followed by a jump to the continuation of the caller.

Exceptions, which can also jump out of a procedure, do not need special treatment since procedure integration does not change the order in which the handlers are pushed.

6.5.5 Specifics of Alice

Due to separate compilation and dynamic linking, the static Alice compiler cannot capitalize on procedure integration. It is much more effective to apply this optimization in the just-in-time compiler because the module boundaries are resolved at run-time and there is more information about immediate procedures.

The analysis phase runs through the abstract code graph. Modifying this graph is incorrect. As we saw in Section 3.1 the graph defines the platform independent code representation. Suppose we integrate a procedure in the first run and store the resulting graph. If, in the second run, the implementation of the procedure changes, the modified code will be invalid. There are three possible solutions: (1) copy the code graph and modify the copied version; (2) modify the original graph, but provide means to switch back to the original one; (3) store information in an external data structure and

[1] In the current framework, the conversion of globals to immediate values is done after the analysis because the call graph is only investigated once, and not for each specialized version. This means that procedure integration itself does not benefit from the specialization.

perform the integration in the compiler. The third alternative was chosen for the byte code system because it requires no complicated transformation and is space efficient. Furthermore, it clearly separates the analysis and transformation phase.

6.6 Self Calls

Almost every interesting computation in Alice is done with recursive procedures. There are no loops in the abstract code. The `while`-loop that the source language provides is only syntactic sugar and is transformed into a recursive procedure. If a procedure is directly recursive, i.e. it invokes itself to compute the next step, we say that the procedure contains a *self call*. The compiler detects self calls during the analysis of immediate procedures (see Section 6.3) by testing if the callee is equal to the caller.

We differentiate between *normal self calls*, which are self calls in non-tail position, and *self tail calls*. The rest of this section introduces specialized byte code instructions for both kinds of self calls.

Specialized Instructions

For a normal self call, the byte code interpreter knows that the callee has a byte code representation and that it is not a transient. This saves two dynamic tests. The special byte code instruction for normal self calls puts all arguments into scheduler registers, issues the creation of a new task, stores the return address and checks for preemption. If the thread is allowed to continue, the interpreter processes the new task. So the specialized instruction saves the indirection over the scheduler and it keeps the state pointers BP, CP, and IP.

For self tail calls, the interpreter need not push a new frame. After the arguments have been passed, control can jump back to the beginning of current code. Self tail calls are thus converted into loops. The semantics of Alice requires a preemption test for the loop to ensure fair distribution of processor time among all live threads. Parameter passing could be realized by using the scheduler registers. Fortunately, we already developed a mechanism to do this more elegantly. We can reuse in-place parameter passing (Section 6.5.2). This has the great advantage that we do not have to pass arguments at all for all registers that are only passed to themselves.

The treatment of self tail calls is suboptimal regarding global and immediate values. The load instructions of these values should be hoisted out of the loop to save reloading in each loop iteration. On the other hand, it would require an additional analysis and a special treatment of scratch registers, whose scope currently comprises only byte code blocks that correspond to abstract code instructions.

7 Implementation

The Alice virtual machine is implemented in C++. On the one hand, C++ offers strong abstraction facilities, for instance, classes, inheritance, and templates. On the other hand, the language still supports low-level features for efficient implementation of virtual machine services. Since C++ allows low-level programming, the programmer can easily introduce errors. For this project, a well-structured development process, which enabled continuous testing, helped to achieve a reliable implementation in the specified time period.

The first section outlines the development process. The second section deals with implementation techniques for efficient interpreters. A section on the implementation of the byte code compiler concludes the chapter.

7.1 Development Process

Building an interpreter together with a byte code compiler is not as complex as a native code compiler, especially if the source language resembles the target language. Nevertheless, it is still an error prone task and finding the cause of an error can be extremely difficult and time-consuming. The implementation strategy for this thesis project aimed to find programming errors as early as possible. As two working execution units have existed already, the SEAM task model could be used for continuous testing during the development process.

Two main components were to implement:

- an efficient *interpreter* to execute byte code
- a *compiler* to transform abstract code into byte code

One cannot test the interpreter implementation as long as there are no means to generate byte code. A compiler offers this functionality, but it cannot be tested as long as there is no interpreter. One solution to break this cyclic dependency is to write the two units in parallel. However, this has some disadvantages. If an error occurs, both compiler and interpreter have to be checked because they are both in the process of development and therefore unreliable.

A better approach is to use a byte code assembler to generate test sequences for the interpreter. As soon as the interpreter is fully implemented and tested, the compiler can be written and tested using the reliable interpreter.

Byte Code (Dis-) Assembler

The byte code *assembler* is integrated into the Alice library. One part is implemented in Alice, and the other part, which needs access to low-level services of the virtual machine, is implemented in C++. SEAM's foreign function interface connects the C++ part with the Alice implementation.

The assembler receives a list of data type items, representing byte code instructions, and transforms the items into the internal byte code representation.

```
datatype register = R0 | R1 | R2 | ...
datatype byte_code_instr =
        iadd of register * register * register
      | ...

val sequence = [
      ccc1 R0,
      load_int (R1,1),
      iadd (R2,R0,R1),
      seam_return R2
   ]
val inc : int -> int = assemble sequence
```

The example implements an increment procedure in byte code. Since the byte code is defined as a data type, the assembler need not parse the input.

To make byte code programming more convenient and independent from the internal representation, the assembly language is extended with string labels that allow a natural formulation of jump instructions:

```
val sequence = [... label "start", ..., jump "start", ... ]
```

The assembler is easy to implement because it is mainly a one-to-one mapping from data type values to the internal byte code representation. However, even in trivial programming tasks, there might be errors. To find bugs in the assembler, we implemented a *disassembler* that transforms the internal byte code representation into a readable string. This way, the code that the assembler generates can be validated.

The assembler is not only useful for debugging. It can also be used to benchmark handwritten byte code against automatically generated code. Thus, one can estimate how worthy a specific compiler optimization will be. For instance, the overhead of superfluous load instructions that simulate parameter passing for integrated procedures (Section 6.5.3) was analyzed in this way.

Interpreter

The interpreter is a loop with a huge case distinction over all instructions. All instructions are independent from each other and can thus be implemented separately. Each instruction is validated with a handwritten test case. At the end, this gives a reliable interpreter and a test suite for regression testing.

Compiler

The compiler translates abstract code instructions to byte code sequences. As soon as the compiler infrastructure is working, code generation can be implemented separately for each abstract code instruction. Writing test cases for the code generator is not as convenient as for the interpreter because there does not exist an assembler for abstract code. However, one can write small Alice procedures that the static compiler translates into the desired abstract code instructions. The generated byte code is tested on the byte code interpreter.

Following these steps cuts the overall task into manageable pieces and finally leads to a reliable implementation. Of course, the first prototype nevertheless contained some errors; but it was only a matter of two days to fix these bugs.

7.2 Interpreter

The main part of the interpreter is a loop with a big case distinction over all instructions. The loop fetches byte code instructions out of the code buffer, checks the opcode and jumps to the implementation to execute it. There are several ways to implement the dispatch loop efficiently. This section restricts itself to two widespread implementation techniques, called *switch-based* and *direct threaded* interpretation.

7.2.1 Switch-based Dispatch

In a switch-based implementation, the byte code is an array of integers that represent both the opcodes and the arguments. The program counter is an index into the byte code array. The dispatch loop contains a C++ `switch` statement with one `case` label per byte code instruction. A small interpreter is exemplified in Figure 7.1.

```
int program[] =
      { iadd, r0, r4, r7, seam_call, r2 , r0, ... };
int pc = 0;

for(;;) {
  switch(program[pc++]) {
    case iadd:
      int r0 = program[pc++];
      int r1 = program[pc++];
      int r2 = program[pc++];
      ...
      break;
   case seam_call:
      ...
      break;
  }
}
```

Figure 7.1: switch-based interpreter

An optimizing C++ compiler usually implements the `switch` statement with a jump table. At run-time, the interpreter fetches the opcode, checks if there is a case label for it (range check), jumps to the implementation, executes it, and jumps back to the `switch` when it reaches `break`. So there is a range check, an indirect and a direct jump. The range check is superfluous as a correct program only consists of known opcodes. Additionally, the direct jump can be eliminated, which brings us to *direct threaded* dispatch.

7.2.2 Direct Threaded Dispatch

In a direct threaded [9] interpreter, the opcodes represent the start addresses of their implementations. The interpreter routine can reach them with an indirect jump. ANSI C does not support this form of jumps. However, the GNU C compiler offers an extension for a platform-independent implementation of direct threaded dispatch. Figure 7.2 shows a part of the source code. The interpreter performs indirect jumps from one instruction to the next. The program counter is just a pointer into the byte code array.

```
int program[] = {
    (int)(&&iadd), r0, r4, r7,
    (int)(&&seam_call), r2 , r0, ...
  };
int *pc = program;

// jump to first instruction
goto *((void *)(*pc++));

iadd:
   // fetch arguments
   int r0 = *pc++;
   int r1 = *pc++;
   int r3 = *pc++;
   ...
   // goto next instruction
   goto *((void *)(*pc++));

seam_call:
  ...
  goto *((void *)(*pc++));
```

Figure 7.2: direct threaded interpreter

By using macros and conditional compilation, the interpreter implementation can be changed from threaded to switch dispatch and back. Direct threaded implementations are usually faster than the switch-based interpreters. For example, the Ackermann function runs 44% faster when direct threaded interpretation is enabled.

7.2.3 Advanced Interpretation Techniques

In principle, code for threaded or switch interpreters can be seen as a list of indirect jump instructions together with some arguments. Modern processors try to predict the destination of an indirect jump. If the prediction is good, this gives a significant speedup. On the other hand, it takes several processor cycles to recover from a false prediction. Unfortunately, the heuristics that are used do not work well for interpreter routines. Consider the following example:

```
iadd R4 R1 R0            (* R4 <- R1 + R0 *)
iadd R5 R2 R3            (* R5 <- R2 + R3 *)
new_pair_init R6 R4 R5   (* R6 <- (R4,R5) *)
```

After executing the first line, the branch prediction unit assumes that `iadd` is most likely followed by itself. This assumption fails for the second `iadd` instruction that is followed by `new_pair_init`.

The heuristic, which works well for other programs, often fails for virtual machine interpreters because it is not the native PC but the virtual PC that correlates with the control flow of the byte code program. There are several techniques to improve branch prediction and to reduce the dispatch overhead of indirect branches.

- Piumarta and Riccardi [33] propose to concatenate the native code chunks that implement an instruction, on basic blocks to eliminate jump instructions. They call their technique *selective inlining*.

- Ertl and Gregg [19] investigated the failure prediction rate for interpreters on modern processors. They proposed several techniques to make the prediction more accurate: static and dynamic super-instructions, instruction replication, and extensions of inlining over basic blocks.
- A recent work uses so called context threading [10] to make better use of the branch prediction hardware units. Berndl et al. use native function calls and can thus capitalizing on branch prediction for calls and returns. They argue that this gives better prediction rates than indirect jumps and report a speed-up of 25% for a Java interpreter.

As presented in Chapter 4.4, the byte code system only uses static super-instructions. Replication does not seem to make sense as it significantly blows up interpreter size and slows down build time[1]. Full inlining of byte code is in its essence a naive way of run-time compilation to native code. In summary, all advanced dynamic approaches depend on the underlying hardware and are not portable.

7.3 Compiler

The compiler traverses the abstract code graph and generates byte code, either for a switch-based or a direct threaded interpreter. This section shows how the graph traversal is implemented and how the compiler can be abstracted over the internal code representation.

7.3.1 Code Traversal

Graph traversal is typically implemented either with an explicit control stack or with recursion. The explicit control stack is supposed to be more efficient in C++. The stack might, for instance, contain the nodes that the compiler has to translate. The maximal stack height is then the number of nodes in the abstract code graph. Recursion needs more stack space since the control stack is implicitly maintained in the procedure call stack. Furthermore, there is one function call per node. However, as shown in Chapter 5, the compiler can be formulated quite naturally with a recursive translation function. Therefore, we decided to implement the byte code compiler based on recursion. The performance penalty is low since the compiler is not as performance critical as the interpreter. High-level optimizations like selective compilation (Section 6.2) save a lot more compilation time than sophisticated implementation techniques.

7.3.2 Code Layout

The final phase of a compiler is the generation of target code that is, for instance, some sort of assembly code or byte code. The native code compiler of Alice uses *GNU lightning* to generate executable native code. Other compilers pass the assembly code to an assembler for further processing to binary code. The byte code compiler also separates code generation from the transformation to the internal representation. As an additional abstraction layer, there is a set of macro definitions to encode byte code instructions into the code buffer and to decode them from the buffer.

[1] Ertl and Gregg report about a Forth interpreter that is 400 MB huge and that needs 5 hours to be compiled.

Byte code resides in a array of words. The current implementation uses one slot for each opcode and one slot per argument. Modern machine architectures are designed for fast access on word boundaries. Experiments in the byte code framework show that a more compact representation reduces code size, but slows down execution. For instance, when two registers are aligned into one slot, the byte code of the Ackermann function is 23% smaller and execution slows down by 2%. As byte code is already small in comparison to native code (see Chapter 8), we prefer the word-sized representation, which is both faster and easier to implement.

Direct Threaded Code

In direct threaded code, the opcodes are addresses of locations inside the interpreter main routine. The GNU C compiler does not allow to access these addresses from outside of the routine. However, the byte code compiler needs the addresses for code generation. The solution is that the interpreter routine is called at startup time of the system with a designated argument (a_{init}) and fills all internal addresses into a global table. This way, the compiler, or more precisely, the layout macros get access to a mapping from instruction numbers to code addresses. The disadvantage of this approach is that the interpreter routine has to check for a_{init} whenever it is called. Fortunately, the test is only a pointer comparison, which causes almost no overhead in practice.

8 Evaluation

In this chapter, we evaluate the byte code system and investigate the effectiveness of different optimizations. All benchmarks were carried out on a 2.8GHz Pentium IV with 1GB of RAM, running Linux 2.4.20.

Abbreviations

We use the following abbreviations throughout the chapter to describe different system configurations:

- *AC:* abstract code interpreter
- *BC:* byte code system with all optimizations; compilation before the second execution. Several different variations of BC are used:

BC^{-pi}	Procedure integration disabled
BC^{-ra}	Register allocation disabled
BC_{lazy}	Compilation before the first execution
BC_{plain}	Base system presented in Chapter 4 and 5 without optimizations
BC_{plain}^{dyn}	Base system extended with dynamic tests in (tail-) call and return instructions
BC_{plain}^{stat}	Base system with compile-time call optimizations, i.e. the compiler differentiates between byte code, immediate, rewrite, and self calls; there are dynamic tests in the `return` instruction
$BC_{plain}^{dyn+stat}$	Base system with compile-time and dynamic call optimizations

- *NC:* native code system with lazy compilation, i.e. every procedure is compiled before its first execution

Interpreting the Benchmark Results

Except for the run-times, all results are given in absolute numbers. For the run-times, the byte code system is taken as a reference point and the relative improvement compared to the other systems is given in percent. So if the table lists 10% for NC, this means that the byte code system is 10% faster than NC, whereas -10% means that BC is 10% slower. A percentage of 300 means that *BC* is 4 times faster.

8.1 System Evaluation

Three kinds of benchmarks were carried out to compare the byte code system to the abstract code interpreter and the native code system. The first kind of benchmarks investigates code size and compilation time of the byte code and native code compiler. The second class of benchmarks are standard performance tests, like the Ackermann function and Fibonacci function. These tests can easily be ported and *MoscowML* [36] is taken for an external comparison. The mosmlc compiler statically compiles the source language to *Caml* [14] byte code, which is executed by the *Caml* byte code interpreter. The third class compares the run-times of three Alice applications, namely the Toplevel interpreter startup, the Alice test suite and the bootstrap process of Alice.

8.1.1 System Characteristics

This benchmark measures code size and compilation time of the Toplevel startup. The Alice Toplevel interpreter provides a prompt that allows entering and evaluating programs in an interactive way. Instead of BC, the system BC_{lazy} is used because it has the same compilation strategy as NC.

mode	compilation time (ms)	code size (MB)	#compiled procs
NC	220	16.64	4361
BC_{lazy}	185	3.35	3773

Table 8.1: compilation time and code size for the Toplevel startup

The byte code compiler is faster although it performs two passes over the abstract code: the first pass is for integration analysis and the second pass is for code generation. As both compilers have a similar structure and use the same compilation strategy, this difference does not come from clever implementation techniques. The main reason for the slower compilation time of the native code system is the massive amount of code it produces. The generated byte code is about 5 times smaller. Therefore, code generation is significantly faster. The difference in the number of compiled procedures stems from procedure integration. So 13.5% of all procedures do not need to be compiled separately as they can always be integrated.

8.1.2 Micro-Benchmarks

By micro-benchmark, we mean the measurement of the run-time of a single (or only a few) procedure(s). There are several procedures that are traditionally used to evaluate the execution speed of a system. Such comparisons can, for instance, be found in "The Computer Language Shootout Benchmarks" [28]. We distinguish three groups of benchmarks.

(1) Recursion and Arithmetic

The Ackermann function, the Takeuchi function, and the Fibonnaci function are examples of functions that heavily use recursion. In each recursive step, they perform some simple integer arithmetic.

(2) Recursion and Symbolic Computation

This group comprises a tail-recursive implementation of `list reverse` and `mergesort` for lists. Both benchmarks test recursion and perform symbolic operations on lists in each recursive step.

(3) Special Feature Benchmarks

One main feature of the byte code system is procedure integration. Table 8.2 shows three benchmarks that particularly benefit from this feature. `loop` is a tail-recursive procedure that increments a counter on each invocation. The counter is a reference cell that is modified with a small helper procedure that the byte code compiler integrates into the loop. `bubblesort` contains the library procedure `Array.swap` that is integrated, too. The benchmark `tree` performs a depth-first-search on a balanced binary tree, using a stack. The stack is implemented in a simple Alice structure and each of its procedures can be integrated.

Results

The run-times represent the arithmetic mean of 20 runs for each benchmark.

	(1)			(2)	
mode	**fib**	**tak**	**ack**	**reverse**	**mergesort**
BC	345	2493	3687	164	1162
AC	461%	514%	743%	429%	287%
NC	−219%	−88%	−303%	−110%	−58%
MoscowML	−217%	−42%	63%	45%	10%

	(3)		
mode	**loop**	**bubblesort**	**tree**
BC	699	2050	1181
AC	2253%	501%	1067%
NC	21%	38%	25%
MoscowML	125%	99%	−2%

Table 8.2: results of micro-benchmark

Table 8.2 shows the benchmark results for (1) and (2). There are massive improvements in comparison to the abstract code system, which formerly was the only platform-independent execution mode. The native code system is fastest on all benchmarks. The difference is most significant for benchmarks that heavily use arithmetic. MoscowML performs well for `fib` and `tak`, but on the other benchmarks, it is slower than the Alice byte code system. MoscowML has efficient arithmetic operations, but symbolic computation seems to be less efficient. The reason for the bad performance of `ack` remains unclear[1].

Table 8.2 also shows the benchmark results for (3). The byte code system massively outperforms the abstract code interpreter and is on average 28% faster than the native code system. The behavior of MoscowML is again a bit mysterious. Although it is two times slower for `loop` and `bubblesort`, it slightly outperforms the Alice byte code system for `tree`. Our conclusion is that it is difficult to obtain a fair comparison between different programming systems because each system has its own characteristics.

[1] Using the *OCaml* byte code compiler delivers the same results. So the performance decrease for `ack` seems to be a peculiarity of the interpreter.

8.1.3 Alice Applications

The previous section presented some micro-benchmarks. These performance tests only evaluate specific features and the results cannot be generalized. Three larger Alice applications are measured to obtain a more realistic comparison between the different execution modes:

- *Toplevel startup:* The Toplevel interpreter provides a prompt that allows entering and evaluating programs in an interactive way. The benchmark measures the startup time of the Toplevel.
- *library test suite:* The Alice test suite checks the correctness and the performance of all basic modules in the Alice library.
- *bootstrap:* Alice is able to compile itself. The bootstrap process generates the run-time system, the compiler, and the Alice library.

Results

mode	Toplevel startup	library tests	bootstrap
BC	1654 ms	10 min : 44 sec	20 min : 50 sec
AC	156%	248%	179%
NC	12%	−11%	0.2%

Table 8.3: benchmark results for different Alice applications

Table 8.3 shows the benchmark results. The native code system is fastest for the library tests. However, the distance to the byte code system is significantly less than in the micro-benchmarks. The Toplevel starts up fastest with BC. For the bootstrap, NC and BC take equally long. The byte code system is even some seconds faster than the native code system. The results confirm our hypothesis that an optimized byte code system can be faster than a simple native code system.

8.2 Evaluation of Optimizations

This section takes a look at some optimizations that the byte code compiler and interpreter apply. Measurements are given to confirm that the optimizations really improve performance. The structure of the section follows the previous one.

8.2.1 System Characteristics

Table 8.4 shows how procedure integration and selective compilation affect compilation time and code size for the Toplevel startup. The reference system is the fully optimized system BC_{lazy} that compiles each procedure on the first execution.

Results

Procedure integration nearly doubles compilation time because an additional pass is needed for the analysis of integration candidates. About 18% more code is generated since procedure bodies may be copied several times into the caller. Interestingly, this does not affect execution speed. Despite the slower compilation and the bigger code,

mode	compilation time (ms)	code size (MB)	#compiled procs
BC_{lazy}	185	3.35	3773
BC_{lazy}^{-pi}	100	2.74	4361
BC	90	1.15	2848
BC^{-pi}	56	1.04	3096

Table 8.4: compilation times and code size for the Toplevel startup

we will later see that the Toplevel startup is 4% faster when procedure integration is used.

The system BC compiles each procedure before the second execution, using selective compilation. The results for compilation time and code size reveal interesting system properties. The numbers show that nearly 25% of all procedures are executed only once. These procedures are big because they amount to 66% of the code size. Additionally, they seem to contain many integration candidates since selective compilation reduces the code growth that comes from procedure integration. In summary, the results indicate that the Alice system contains some huge procedures that are better not compiled since they are only executed once.

8.2.2 Micro-Benchmarks

All micro-benchmarks were tested with different configurations of BC.

Results

The results of the benchmarks are depicted in Table 8.5. The numbers for micro-benchmarks of kind (1) and (2), and for (3) are summarized to obtain a more compact presentation.

mode	$\sum(\mathbf{1}) + \sum(\mathbf{2})$	$\sum(\mathbf{3})$	lib test	Toplevel
BC	7888 ms	3980 ms	10 min : 44 sec	1654 ms
AC	577%	969%	248%	156%
BC_{plain}	286%	555%	95%	64%
BC_{plain}^{dyn}	213%	445%	77%	53%
BC_{plain}^{stat}	201%	402%	61%	43%
$BC_{plain}^{stat+dyn}$	191%	377%	56%	36%
BC^{-pi}	0%	146%	5%	4%
BC^{-ra}	5%	29%	0.5%	16%

Table 8.5: benchmarks for different system configurations

The numbers show that optimizations are essential for the performance. Naive transformation from abstract code to byte code results in a system that is more than 4 times slower than optimized byte code. However, BC_{plain} is already twice as fast as the abstract code interpreter.

We pointed out that improvements for procedure calls are important optimizations. The byte code system uses two different approaches that complement each other. The benchmarks show that dynamic tests improve performance by 70%. The compile-time approach is slightly more effective, and the best idea is to use both approaches in combination.

The system BC^{-pi} shows equal performance for benchmarks of kind (1) and (2) since they cannot benefit from procedure integration. Benchmarks of kind (3) are explicitly tuned to benefit from the feature. They are 2.5 times faster when procedure integration is enabled.

Section 6.4 showed that register allocation reduces memory demands. The question is whether this affects execution speed, and the answer is yes. The standard benchmarks run 5% faster. The benchmarks whose performance depends on procedure integration speed up by 29%. The reason is that procedure integration depends on register allocation to eliminate superfluous load instructions (see Section 6.5.3).

8.2.3 Alice Applications

The Toplevel interpreter and the Alice test suite are taken to benchmark real applications. Table 8.5 shows the results. Performance differences for different optimizations are not as extreme as for the micro-benchmarks, but the trend is the same.

The simple byte code system BC_{plain} is already up to 150% faster than the abstract code interpreter. Compile-time call optimizations are slightly more effective than dynamic tests, and both approaches complement each other.

Procedure integration speeds up the benchmarks about 5%. The numbers for register allocation depend on the benchmark. Whereas the Toplevel startup is 16% faster when register allocation is enabled, the performance increase for the library test is negligible.

8.3 Summary of all Results

The benchmark results show that the performance of the byte code system comes close to the native code system. For some applications, like the Toplevel startup and the bootstrap, the byte code system is even faster. The byte code system massively outperforms the abstract code interpreter and has therefore become the standard platform-independent execution unit in Alice.

The performance of the byte code system heavily depends on optimizations. Dynamic tests and compile-time specialization double execution speed. For procedure integration, the picture is mixed. We presented realistic micro-benchmarks that run more than twice as fast when procedure integration is enabled, whereas real applications moderately speed up by 5%. In summary, this chapter proved that the optimizations really speed up execution.

9 Conclusion

This chapter concludes the thesis by summarizing the main contributions. Some topics for future work on the Alice run-time system are proposed at the end.

9.1 Summary

Alice programs are executed by a virtual machine, which is implemented on top of the generic virtual machine framework SEAM. The framework explicitly allows mixed-mode execution. This thesis developed a byte code system, which serves for efficient platform-independent execution of Alice programs. The project comprised three main parts.

(1) Specification of an Alice Byte Code

The Alice byte code is derived from the abstract code, which is the external storage format for Alice code. There is at least one byte code instruction for each abstract code instruction. The byte code instructions only keep information (in form of arguments) that is needed during execution. Additional information, like debug annotations, are removed. Some abstract code instructions, which perform multiple basic execution steps, are split into several byte code instructions. For performance reasons, there are many byte code instructions for frequent special cases. To eliminate expensive procedure calls, there are special instructions for frequent primitives, like integer arithmetic operations. In contrast to the abstract code, the byte code is linear and the control flow is simulated with jump instructions.

(2) Just-In-Time Byte Code Compiler

The static Alice compiler transforms the source program into a set of abstract code procedures. The byte code compiler is in charge of transforming the abstract code graphs to byte code chunks at run-time. Optimizations are essential to achieve performance that is competitive to the native code system. Call optimizations and instruction specialization are lightweight techniques that are highly effective to speed up execution. Additionally, the byte code compiler features loop transformation with elimination of parameter passing, procedure integration, and a more flexible mechanism to trigger compilation, called *selective compilation*.

(3) Byte Code Interpreter

The byte code is executed by an interpreter that is embedded into the virtual machine as a task manager for byte code. The byte code interpreter uses direct threaded interpretation, which is both fast and portable. Additionally, it features several optimizations to avoid indirections over SEAM's scheduler, including dynamic tests and dynamic code rewriting.

The performance evaluation shows that the byte code system is often more than 4 times faster than the abstract code interpreter. So it noticeably improves platform-independent execution speed. For larger applications, the performance is even competitive to the native code system, and it outperforms the native code system for the bootstrap and the Toplevel startup.

9.2 Future Work

This section proposes ideas for future projects to improve the Alice virtual machine. The main focus lies on execution speed. We also discuss some design decisions that could have been made differently.

Strictness Analysis

Alice supports lazy and concurrent programming. Both concepts are based on the notion of transients. A transient is a placeholder for an undetermined value. To bind a transient to an actual value, the transient is transformed into a reference that points to the actual value. Since the actual value might again be a transient, reference chains can become arbitrarily long. Instructions that require actual values always have to check for transients and dereference the chains. To remove superfluous testing and dereferencing, a compiler could perform a kind of strictness analysis. The simple strictness analyzer for Haskell by Jones and Partain [27] could serve as a first basis. However, transients are more general than lazy values. "Touch Optimization" presented by Flanagan and Felleisen [20] should also be considered.

Since the additional compiler phase is likely to be expensive, it is reasonable to perform (at least parts of) the analysis in the static compiler. Rough measurements in the byte code interpreter suggest that in more than 99% of all cases there is no request needed. These results neither reflect how many tests a compile time analysis can remove, nor do they reveal anything about superfluous dereferencing. However, they suggest that the run-time system surely benefits from strictness analysis and touch optimization.

Selective Compilation

The current byte code system only uses the basic functionality of selective compilation. It would be interesting to experiment with different code stages. For example, the system could start by interpreting the abstract code. After several executions, a slightly optimizing byte code compiler could generate byte code. For heavily recursive procedures, the system could finally generate highly optimized byte code or native code.

It would also be interesting to consider more sophisticated heuristics for the transitions between code stages. The counter heuristic proved to be effective in most cases. However, the bootstrap time slightly decreases when a lazy compilation strategy is used. A more fine-grained heuristic may be profitable. To get a better heuristic, one could introduce counter decay, which means that the counters are periodically divided by two ([42], p. 165). This approach accommodates the relative frequency of calls. So recursive procedures are compiled early, whereas procedures that are only called a few times per second are not modified.

Procedure Integration

There is few literature on technical aspects of procedure integration. Jones and Marlow encountered this lack and present detailed information about the *Glasgow Haskell Inliner* [26]. In this thesis, we pointed out several technical difficulties and pitfalls of procedure integration in Alice. We decided to clearly separate the analysis from code generation and not to modify the abstract code graph during the analysis. This approach has the advantage that services like pickling remain unchanged. An alternative approach is to modify the abstract code, such that the compiler does not have to know about procedure integration. This way, the optimization could more easily be incorporated into other compilers, like the native code compiler. On the other hand, backwards transformation to the original abstract code would be necessary in order to enable pickling.

The main criterion to decide what procedure to integrate is code size. One can think of several extensions to this simple heuristic. For instance, bigger procedures could also be integrated if the caller invokes them with a high frequency.

Procedure integration could serve as a starting point for recursion unrolling because both are technically very similar. Rugina and Rinard investigate "Recursion Unrolling for Divide and Conquer Programs" [39] and report that the resulting programs run up to 10 times faster.

We already indicated that the system benefits from procedure integration not only because the procedure call overhead is removed, but also because there are more immediate values available to the compiler. One could investigate constant propagation and folding. For instance, if an actual argument is an immediate tagged value, all tests on this values inside the integrated callee can be removed. Some nested tag-tests could also be simplified if the tests of the caller and the callee overlap. We already collected some rough measurements for tagged values and found out that one can skip around 40 tag-tests during the Toplevel startup. However, a more careful analysis and implementation is required for definite statements about the power of constant propagation.

Hoisting of Global Variables and Immediate Values

Abstract code instruction can directly operate on global variables and immediate values. The byte code interpreter loads them into scratch registers in order to make them accessible to byte code instructions that only operate on registers. Since scratch registers are only alive in the code block that stems from the compilation of a single abstract code instruction, an immediate value or global variable that is used more than once in several consecutive abstract code instructions is loaded several times.

To remove superfluous load instructions, hoisting of global variables and immediate values could be used. They could be hoisted in the abstract code graph and a `Let` instruction could be introduced to move them into local identifiers. This way, they would be included into the liveness analysis and register allocation. Experiments show that in many cases there are at most two load instructions to the same value on a path. So it remains unclear whether the minor improvement in code quality justifies modifications of the abstract code format and additional analysis overhead.

Byte Code Debugger

The byte code system offers a simple disassembler. In combination with profiling code added by hand to the interpreter, this was sufficient for debugging. However, a more

general and more powerful debugging facility is preferable. During the implementation of new concepts, the interpreter crashes often because some virtual registers contain wrong values. C++ debuggers only reveal which byte code instruction crashed, but to find out the reason for the crash, step-by-step evaluation with interactive inspection of the interpreter state would be very helpful. A debugger would simplify maintainability of the system.

Improvements to the Static Compiler

The static compiler could provide some additional annotations to guide just-in-time compilation.

- The number of nodes in the abstract code graph could easily be computed during static compilation. This would eliminate counting in the procedure integration phase.
- Recursive calls should be marked. This would save some hacks in the run-time compiler.
- Alice functors could be marked. Then, as an experiment, one could specialize functors to their arguments. These arguments are often procedures for which procedure integration could be used.

Many compiler optimizations, for instance the call optimizations, only work for immediate values. The static compiler uses closure specialization (see Section 6.3.2) to enable the run-time compiler to transform global variables to immediate values. However, the heuristic that decides whether to generate a specialized closure or not is rather ad-hoc. Every closure that is declared top-level is specialized. This captures all procedures of a module that are visible from outside. More sophisticated techniques are desirable to capture procedures that are declared locally.

We already mentioned that cross-component optimization in Alice may statically be impossible. However, it would be very interesting to investigate partial analyses. This means that the static compiler takes the imports as preconditions and tries to derive some invariants from them; or it creates a general form of code templates that the just-in-time compiler can instantiate during compilation.

10 Bibliography

[1] The Java HotSpot Virtual Machine, v1.4.1. Technical white paper, SUN microsystems, September 2002.

[2] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[3] The Alice Project. `http://www.ps.uni-sb.de/alice`, 2006. Programming Systems Lab, Universität des Saarlandes, Saarbrücken.

[4] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th symposium on Principles of programming languages*, pages 1–11. ACM Press, 1988.

[5] U. Ammann. On code generation in a Pascal compiler. *Software – Practice and Experience*, 7(3):391–423, 1977.

[6] A. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[8] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[9] J. Bell. Threaded code. *Communications of the ACM*, 16:370–372, 1973.

[10] M. Berndl, B. Vitale, M. Zaleski, and A. Demke Brown. Context Threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization*, 2005.

[11] G. Bilardi and K. Pingali. Algorithms for computing static single assignment form. *Journal of the ACM*, 50(3):375–425, May 2003.

[12] P. Bonzini. GNU Lightning. `http://www.gnu.org/software/lightning/lightning.html`, 2006.

[13] T. Brunklaus and L. Kornstaedt. A virtual machine for multi-language execution. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, 2002.

[14] The Caml Language. `http://caml.inria.fr`, 2006. INRIA.

[15] G. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[16] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[17] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Proceedings of the ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49. ACM Press, 2003.

[18] A. Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 315–327. ACM Press, 1995.

[19] A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 278–288. ACM Press, 2003.

[20] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *POPL '95: Proceedings of the 22nd symposium on Principles of programming languages*, pages 209–220. ACM Press, 1995.

[21] J. Gosling, G. Steele, G. Bracha, and B. Joy. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[22] R. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[23] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[24] R. Ierusalimschy, L. de Figueiredo, and W. Celes. The Implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005.

[25] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128. ACM Press, 1999.

[26] S. Jones and S. Marlow. Secrets of the Glasgow Haskell compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.

[27] S. Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In *Functional Programming Glasgow, Workshops in Computing*, pages 201–220, 1993.

[28] The Computer Language Shootout Benchmarks. `http://shootout.alioth.debian.org`, 2006.

[29] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report RT-0117, INRIA, February 1990.

[30] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[31] Glenford J. Myers. The case against stack-oriented instruction sets. *SIGARCH Comput. Archit. News*, 6(3):7–10, 1977.

[32] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. In *Proceedings of the 5th International Workshop on Frontiers in Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 248–263. Springer, August 2005.

[33] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 291–300. ACM Press, 1998.

[34] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Sys.*, 21(5):895–913, 1999.

[35] B. Randell and L. Russel. *ALGOL 60 Implementation. The Translation and Use of Algol, 60 Programs on a Computer*, volume 5 of *A. P. I. C. Studies in Data Processing*. Academic Press, 1964.

[36] S. Romanenko, C. Russo, and P. Sestoft. Moscow ML Language Overview. Technical report, 2000.

[37] B. Rosen, M. Wegman, and F. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th symposium on Principles of programming languages*, pages 12–27. ACM Press, 1988.

[38] A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*. Intellect Books, February 2006.

[39] R. Rugina and M. Rinard. Recursion unrolling for divide and conquer programs. In *LCPC '00: Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, pages 34–48. Springer, 2001.

[40] SEAM. `http://www.ps.uni-sb.de/seam`, 2006. Programming Systems Lab, Universität des Saarlandes, Saarbrücken.

[41] Y. Shi, D. Gregg, A. Beatty, and A. Ertl. Virtual machine showdown: stack versus registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 153–163. ACM Press, 2005.

[42] J. Smith and R. Nair. *Virtual Machines*. Morgan Kaufmann, 2005.

[43] G. Tack. Linearisation, minimisation and transformation of data graphs with transients. Diploma thesis, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, May 2003.

[44] G. Tack, L. Kornstaedt, and G. Smolka. Generic pickling and minimization. In *Proceedings of the ACM SIGPLAN Workshop on ML*, volume 148, pages 79–103, March 2005.

[45] J. von Ronne, N. Wang, and M. Franz. Interpreting programs in static single assignment form. In *Proceedings of the ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators*, 2004.

[46] P. Wilson. Uniprocessor garbage collection techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 1–42. Springer, 1992.