Dynamic Compilation for Functional Programs

vorgelegt von Diplom-Informatiker

Martin Grabmüller

aus München

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Sebastian Möller
Berichter:	Prof. Dr. Peter Pepper
Berichter:	Prof. Dr. Sabine Glesner

Tag der wissenschaftlichen Aussprache: 21. April 2009

Berlin 2009

D 83

Zusammenfassung

Diese Arbeit behandelt die dynamische, zur Laufzeit stattfindende Übersetzung und Optimierung funktionaler Programme. Ziel der Optimierung ist die erhöhte Laufzeiteffizient der Programme, die durch die compilergesteuerte Eliminierung von Abstraktionen der Programmiersprache erreicht wird.
Bei der Implementierung objekt-orientierter Programmiersprachen werden bereits seit mehreren Jahrzehnten Compiler-Techniken zur Laufzeit eingesetzt, um objekt-orientierte Programme effizient ausführen zu können. Spätestens seit der Einführung der Programmiersprache Java und ihres auf einer abstrakten Maschine basierenden Ausführungsmodells hat sich die Praktikabilität dieser Implementierungstechnik gezeigt. Viele Eigenschaften moderner Programmiersprachen konnten erst durch den Einsatz dynamischer Transformationstechniken effizient realisiert werden, wie zum Beispiel das dynamische Nachladen von Programmteilen (auch über Netzwerke), Reflection sowie verschiedene Sicherheitslösungen (z.B. *Sandboxing*).
Ziel dieser Arbeit ist zu zeigen, dass rein funktionale Programmiersprachen auf ähnliche Weise effizient implementiert werden können, und sogar Vorteile gegenüber den allgemein eingesetzten objekt-orientierten Sprachen bieten, was die Effizienz, Sicherheit und Korrektheit von Programmen angeht.
Um dieses Ziel zu erreichen, werden in dieser Arbeit Implementierungstechniken entworfen bzw. aus bestehenden Lösungen weiterentwickelt, welche die dynamische Kompilierung und Optimierung funktionaler Programme erlauben: zum einen präsentieren wir eine Programmzwischendarstellung (getypte dynamische Continuation-Passing-Style-Darstellung), welche sich zur dynamischen Kompilierung und Optimierung eignet. Basierend auf dieser Darstellung haben wir eine Erweiterung zur verzögerten und selektiven Codeerzeugung von Programmteilen entwickelt. Der wichtigste Beitrag dieser Arbeit ist die dynamische Spezialisierung zur Eliminierung polymorpher Funktionen und Datenstrukturen, welche die Effizienz funktionaler Programme deutlich steigern kann. Die präsentierten Ergebnisse experimenteller Messungen eines prototypischen Ausführungssystems belegen, dass funktionale Programme effizient dynamisch kompiliert werden können.

Summary

This thesis is about dynamic translation and optimization of functional programs. The goal of the optimization is increased run-time efficiency, which is obtained by compiler-directed elimination of programming language abstractions.

Object-oriented programming languages have been implemented for several decades using run-time compilation techniques. With the introduction of the Java programming language and its virtual machine-based execution model, the practicability of this implementation method for real-world applications has been proved. Many aspects of modern programming languages, such as dynamic loading and linking of code (even across networks), reflection and security solutions (e.g., sandboxing) can be realized efficiently only by using dynamic transformation techniques.

The goal of this work is to show that functional programming languages can be efficiently implemented in a similar way, and that these languages even offer advantages when compared to more common object-oriented languages. Efficiency, security and correctness of programs is easier to ensure in the functional setting.

Towards this goal, we design and develop implementation techniques to enable dynamic compilation and optimization of functional programming languages: we describe an intermediate representation for functional programs (typed dynamic continuation-passing style), which is well suited for dynamic compilation. Based on this representation, we have developed an extension for incremental and selective code generation. The main contribution of this work shows how dynamic specialization of polymorphic functions and data structures can increase the run-time efficiency of functional programs considerably. We present the results of experimental measurements for a prototypical implementation, which prove that functional programs can efficiently be dynamically compiled.

Contents

Chapter 1

Introduction

1.1 Motivation

Modern applications and software systems are no longer monolithic pieces of software. Most systems are designed as a combination of a static core component, and a set of extensions. The core implements the basic functionality and extension infrastructure and the extensions provide additional features. Together, the core and the extensions solve the problems for which the complete system was designed. Examples can be found in operating systems, where file systems, device drivers and user interfaces are built on top of an operating system kernel which manages basic hardware devices like the processor(s) or memory. Text processing systems come with basic editing functionality and can be extended with formula editors, bibliographical databases or multiple output and export formats. Web browsers use plug-ins in order to present multimedia and interactive contents.
There are several advantages in using such an open, extensible system design. The systems are more configurable, because it is easy to integrate customized components into an otherwise off-the-shelf system. As only components needed for a particular task need to be loaded into the system, resource consumption is reduced. Dynamic loading of portable program representations such as bytecode opens the possibility of distributed applications without complicated setup procedures on each node of a network. For long-running systems (such as telecommunication switches or databases), dynamic software updating offers system maintenance without expensive down-times.
The disadvantages of dynamic and open systems lie mainly in their vulnerability (can components from the network be trusted?) and the lower performance due to the necessary abstractions, such as encapsulation and the need to conform to standard interfaces when composing modules. The work presented here is concerned with the reduction of these overheads. Security issues are not a topic of this thesis and are not discussed any further except that our work is in the setting of strongly typed functional languages, which offer some security advantages independently of dynamic compilation.
Object-oriented languages such as Java or C# do support the characteristic features of dynamic languages: dynamic loading and linking, portable representations of pro-

grams and load-time verification for security. The efficiency problem is solved by run-time compilation (also called just-in-time or JIT compilation). Thus run-time compilation seems to be a useful technique for realizing dynamic and flexible systems.
In the world of functional languages, only few attempts have been made to create implementations which are as dynamic and open as their object-oriented counterparts (see Section 2.2.4 for references). Dynamic compilation has been used, for example, for saving memory, but the potential of dynamic optimization in functional languages has not been systematically explored. We plan to go further in that direction.
In the approach we present in this thesis, dynamic compilation is a central implementation characteristic, and all language features we wish to integrate can rely on its presence. All aspects of dynamic systems as outlined above are in principle supported by it.
We advocate an implementation style for functional languages which is heavily influenced by object-oriented implementations. Programs are stored in a portable format. This format can be an efficient encoding of annotated abstract syntax trees where security is enforced by type-checking any intermediate code that is loaded into the system. Programs are therefore easily transferred over networks. The combination of program parts from different sources is handled by the fast dynamic compiler, as well as any necessary specialization due to parametric polymorphism or overloading. Data structures are optimized for efficient memory layout and access, depending on the cache architecture and instruction set of the executing machine.
Functional languages offer an advantage over established object-oriented languages because their powerful type systems can be used to ensure that they do not perform any malicious operations.
Towards our goal, we present in this thesis several techniques for increasing the efficiency of functional programming languages. The first one is used for eliminating language abstractions at run-time by applying an aggressive specialization strategy, which eliminates most of the costs normally associated with these abstractions. Secondly, incremental compilation reducesthe cost of dynamic compilation by limiting it to active program parts and spreading it over the life time of the program. Additionally, we investigate how established optimizations for functional programming languages have to be adapted to fit with our specialization approach.
Future work may build on these foundations in order to implement a complete dynamic execution environment for functional programs.

1.1.1 Problem

The technical possibilities and problems of implementations of dynamic specialization methods have been investigated for several functional programming languages (see Chapter 2 for the relevant literature). Nevertheless, several questions remain open:

– The literature mentions dynamic specialization as a solution to implementation problems, but it has not been widely used, mostly because it was deemed impractical. The technological advances over the last decade let us view this problem in a new light, especially since many dynamic compilation problems have already been solved for object-oriented languages.

- Since there has not been much work on dynamic compilation of functional programs in general, it is not clear which implementation techniques provide a good balance between increased execution efficiency and compilation overhead.
- The techniques which have been shown to be useful for static compilation of functional programs need to be reviewed for their suitability for dynamic compilation.
- Techniques developed for the dynamic compilation of other language paradigms (imperative, object-oriented) should be examined whether they fit the functional paradigm.
- Implementation techniques have always been influenced by language features provided by the implemented languages (e.g., laziness, higher-order functions). We also want to investigate whether programming language design should be influenced by the implementation possibilities provided by dynamic compilation.

In all of these areas, we provide answers, backed up by experimental evidence.

1.1.2 Thesis

The main thesis of this work is the following:

> Functional languages can be efficiently implemented in a dynamically compiling system.

We lay the foundations for this dynamic approach by describing several techniques, on which dynamically optimizing implementations of functional languages can be based. Experiments suggest that our proposed implementation strategy is feasible in practice.

1.2 Contributions

- The main contribution of this work is the presentation of dynamic optimization techniques for functional languages, namely *incremental compilation*, *run-time specialization* of polymorphic functions and data types and *typed dynamic continuation-passing style* as a technique for implementing specialization as well as traditional optimizations. Specialization of polymorphic functions is the most effective technique developed here.
- Additionally, the ideas developed in the theoretical part of this thesis have been implemented in a prototypical system for run-time compilation of functional languages.
- We have measured a performance advantage due to dynamic specialization when compared with other functional-language implementations.

1.3 Outline of this Dissertation

The goal of this dissertation is to present the design and implementation of several techniques for dynamic compilation of functional programs. We have structured the thesis as follows: First, Chapter 2 gives the background about functional languages and their compilation, dynamic (or run-time) compilation, virtual machines, various compilation techniques such as analysis and transformation, continuation-passing style and type-directed compilation. Chapter 3 gives an overview of the design and implementation of the dynamically optimizing system we have implemented as a prototype. The following three chapters go into detail with respect to the new implementation techniques: Chapter 4 presents our approach to dynamic compilation and optimization of functional programs. We present our intermediate language design and the transformation into this language. The use of type information in this program representation and the transformation algorithms is important for the further development. Chapter 5 presents our approach to incremental dynamic compilation, which is based on a small extension to our intermediate representation. Chapter 6 describes the answer to the most important question of this thesis: how can the specialization of polymorphic functions and polymorphic data types be performed efficiently at run-time. Chapter 7 describes our prototype system in detail and shows how classic optimizations have to be adapted for a dynamic system. Chapter 8 gives experimental results of our prototype implementation of dynamic compilation and compares it to other language implementations. Chapter 9 summarizes the work, highlights the main points and describes topics for future work in the area of dynamically compiling functional language implementations.

Chapter 2

Background

In this chapter, we review the basic concepts of dynamic compilation and optimization. We first describe the concepts of functional programming and what dynamic compilation is, give an introduction to virtual machines and give references to related work in this area. We will investigate the application of dynamic compilation techniques to the implementation of programming languages in general and functional languages in particular.

2.1 Functional Programming

The roots of functional programming lie in the 1950s in the development of the programming language Lisp (McCarthy, 1960), which already shared many aspects with today's functional languages: Lisp promotes an expression-oriented style of programming in contrast to the statement-oriented style in imperative languages, in general, Lisp programs have a more mathematical flavor than other languages of its era. Early Lisp implementations already included garbage collectors. Building on these foundations, functional languages like FP (Backus, 1978), Hope (Burstall et al., 1980), ML (Milner et al., 1997), Opal (Pepper, 2003), Clean (Brus et al., 1987), Erlang (Armstrong et al., 1996), Gofer (Jones, 1994) and Haskell (Peyton Jones, 2003) evolved, each developing different and overlapping language features.
The key advantages of functional programming languages are:

Strong typing Many functional languages are strongly typed and provide sophisticated type systems with type inference, polymorphism, uniqueness typing, type classes, parameterized modules, functors, signatures, etc. These make possible the encoding of many program invariants in the source code, increasing the reliability, robustness and maintainability of programs

Modularity Functional programs consist of small, maintainable and isolated functions. Powerful abstraction facilities like functors, parameterized modules or type classes increase modularity further.

Controlled side-effects Purely functional languages strictly separate effect-free expressions from side-effecting actions, enforcing a disciplined programming style. Even impure languages, such as Lisp, Standard ML or Erlang encourage an expression-oriented style by providing syntactic features for the convenient composition and application of functions.

Inherent parallelism Functional programs are in theory well suited for parallelism. The early work in the 1980s on parallel graph reductions has not worked out, but the recent rise of multi-core architectures has renewed interest in this area.

For a detailed introduction to functional programming, many books and tutorials on functional programming are available (Bird and Wadler, 1988; Hudak et al., 1999; Pepper, 2003).

2.2 Dynamic Compilation

A survey on the field of dynamic compilation (or just-in-time compilation, JIT) has been written by Aycock (2003). Aycock's article concentrates on dynamic compilation in software for object-oriented languages. The book by Smith and Nair (2005) on virtual machines also covers many aspects of dynamic compilation as an implementation technique of these machines, both in hardware and software.

The main characteristic of dynamic compilation is that some translation of a program takes place while it is executing. The translation process requires some entity which actually translates the source to the target program. This can be a component of the executing program (for example, in the program's run-time system), part of the operating system or even some program implemented partly or completely in hardware. Modern complex instruction set computers (CISC) translate machine code into a reduced instruction set (RISC) prior to execution. These instructions, called μ-ops in Intel terminology, are held in a special first-level instruction cache (trace buffer).

The motivation of dynamic compilation is increased performance over interpreted systems, while providing the dynamic properties of such systems. Additionally, dynamic compilation may even offer superior performance to statically compiled programs: precompiled binaries often use the least common denominator of the instruction sets of the targeted machines. Newer machines in a processor series often have new instructions with better performance, but programs are normally compiled for older version in order to be compatible. Intel's IA-32 Architecture Optimization Reference Manual (Intel Corporation, 2005) even recommends to use the `cpuid` instruction to determine the processor model and to select code optimized for this model. A dynamically compiling system can use this method without the need to deliver several versions of an algorithm, each optimized for another processor. Other advantages of dynamic compilation include reduction of memory requirements and energy (Wu et al., 2005; Unnikrishnan et al., 2002).

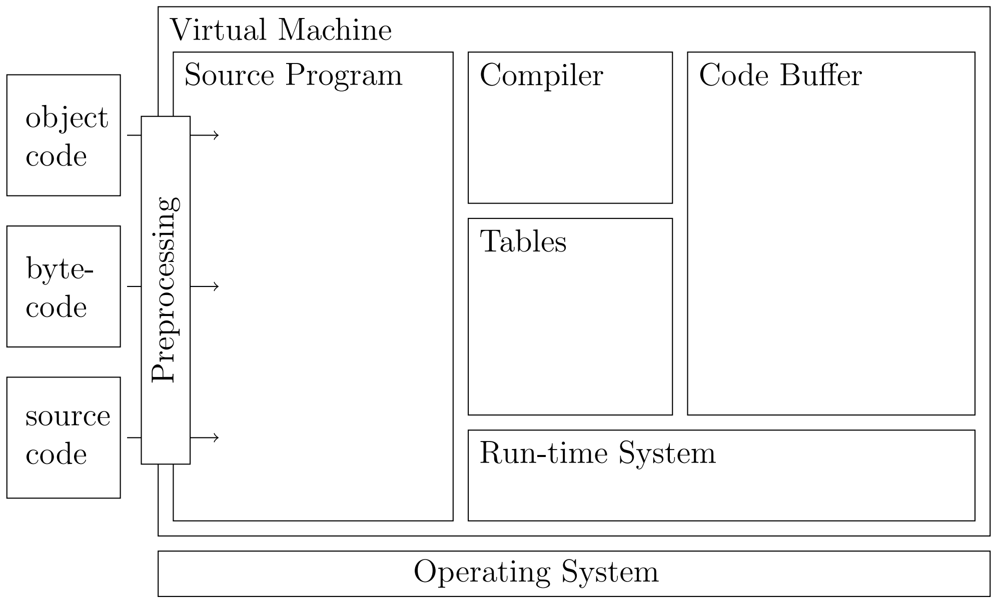

Figure 2.1: Virtual machine

2.2.1 Virtual Machines

A virtual machine, also called an abstract machine, is an idealized model of a machine. It provides a machine architecture consisting of registers, stacks and memory, and an instruction set which manipulates the machine state. On the one hand, virtual machines are used to model language semantics and program evaluation, on the other for the implementation of programming languages on real machines. In some systems, a virtual machine only exists conceptually, for example in the form of an intermediate language of a compiler, in others the virtual machine is a piece of software or hardware, which is capable of executing programs written in its language, thus emulating the virtual machine on real hardware. A prominent example of a software virtual machine is the Java Virtual Machine (JVM) (Lindholm and Yellin, 1999). An example of a hardware virtual machine is the Transmeta Crusoe processor (Klaiber, 2000). The Crusoe processor is a very long instruction word (VLIW) processor capable of executing standard x86 binary code files by translating them into its VLIW instruction set on the fly.

Virtual machines are used to abstract from certain details of real machines. Software virtual machines provide a uniform interface to operating system services (independent of the operating system the machine runs on) and several other services, such as garbage collection or concurrent programming primitives. Hardware virtual machines abstract from the actual hardware implementation by providing an instruction set of another machine.

In the following, we will concentrate on dynamically compiling virtual machines, which have the general structure shown in Figure 2.1.

Dynamically compiling virtual machines possess storage areas for an input program as well as for the actual machine code. The input to such a machine may be source code, some form of compiled portable bytecode or even actual machine code: either for the same architecture (for binary optimizers) or some other architecture (for binary translators). The central component is a dynamic compiler, which translates fragments of the input program into fragments of the target program. The size of the fragments varies from system to system, many operate on the size of individual procedures, others on basic blocks or so-called extended basic blocks (instruction sequences without in-going control-flow edges, but with one or more out-going edges). The compiler needs various data structures to support its operation, and often some dynamic profiling mechanism to keep track of the occurrence of certain events. The collected information is stored in data structures (called tables), which are also part of the system. Finally, a run-time system is needed which provides an interface to the operating system and abstracts from the actual operating system and machine characteristics. When a program is loaded into the virtual machine, often some form of non-trivial preprocessing is necessary, such as initialization of data structures, verification or type-checking of the loaded code. Many virtual machines perform some kind of translation on the code, for example translation into threaded code (Bell, 1973; Dewar, 1975).

In contrast to classic compiler design, the frontier between translation time and run time is not clearly visible in dynamically compiling systems. A dynamically compiling system behaves similar to an interpreted system, where the program to be executed is loaded into a running system which initiates the execution of the loaded program image. In an interpreter, the loaded program is executed one instruction at a time, whereas a dynamically compiling system translates it into machine language before executing it, so that it runs without any interpretative overhead directly on the hardware. This gives a significant performance advantage when compared to an interpretive system, but it is necessary to take the time required for the translation into account. Adaptive systems only use expensive optimizing techniques for those parts of the program which are heavily used (so-called hot spots). The goal is to invest much translation time and space only into parts which use up the majority of the program's actual run time. In order to reach that goal, various heuristics are used, for example the dynamic measurement of program behavior (profiling). The program's execution may be monitored to determine the functions invoked most, or those taking most of the execution time.

Dynamic systems which allow run-time loading of program parts, possibly from untrusted sources, need ways to protect themselves from malicious code. Several techniques have been developed for this task: bytecode verification (as used in Java) performs type checks and some additional integrity checks on the intermediate code when it is loaded. This technique requires an intermediate language which is reasonably easy to check. In the typed assembly language (TAL) approach (Morrisett et al., 1999), concrete machine code is generated and stored in the object code files together with additional type information, which enables the loader to perform the verification. Verification of bytecodes or typed assembly language programs requires building proofs of program safety, whereas in the proof-carrying code approach (Nec-

ula, 1997), programs are annotated with proofs of their correctness, thereby only requiring proof checking. Since proof checking is much easier than finding proofs, this technique simplifies the run-time system considerably. The above safety approaches are orthogonal to the techniques described in this thesis and we will not discuss them in the remainder of the thesis.
One aspect not found in statically compiling or pure interpretive systems is the management of translated code. Normally, only some limited amount of memory is available for storing compiled code fragments in the so-called code cache. When it fills up, some compiled fragments must be removed from the cache so that its space can be re-used. Hazelwood Cettei (2004) investigates several code cache management strategies and measures their effects. Code cache management is also orthogonal to the topics discussed in this thesis and is not treated in the remainder of the text.

2.2.2 History of Dynamic Compilation

Dynamic compilation has been used successfully in virtual machines for object-oriented languages for more than 20 years now.
Smalltalk implementations have used dynamic compilation for the efficient execution of dynamically typed object-oriented languages (Deutsch and Schiffman, 1984; Ogata and Doi, 1994). Since in Smalltalk every operation involves a method invocation, optimization of common operations is very important, and correct implementation is only possible when it is possible to undo optimizations when the compiled system is changed. Implementations of the object-oriented programming language Self (Chambers and Ungar, 1989; Hölzle, 1994) have continued that tradition and added many very sophisticated dynamic optimizations, mostly necessary because Self is even more dynamic than Smalltalk. Java also builds on the techniques developed for Self (Paleczny et al., 2001) and has become a very successful language, mainly due to the portability and efficiency which is made possible by using dynamic compilation techniques. Franz (1994) describes another interesting implementation for the Oberon language (Wirth and Gutknecht, 1998), which uses portable program representations and load-time compilation.
Dynamic compilation techniques in general have been developed even earlier. Brown (1976) describes a Basic system which uses so-called "throw-away compilation", where each Basic statement is compiled prior to execution and the compiled statements are cached. When the memory buffer allocated for compiled code gets full, all compiled code is thrown away, hence the name.
An important – though not widely known or cited – work in the field of dynamic compilation is the Synthesis operating system kernel, developed by Pu et al. (1988). All operating system services in Synthesis rely on dynamic code generation. For example, when a file is opened and this file belongs to a FIFO pipe, special code for writing to and for reading from pipes is generated, when the file is a disk file, other device-specific code is generated. Massalin (1992) showed that the performance of such as system can be an order of magnitude better than other operating systems, thus enabling new features such as very fine-grained threading.

2.2.3 Dynamic Compilation Techniques

Dynamic compilation can be divided into program transformations performed by software, as part of the run-time system of some language implementation, and transformations performed by hardware, as part of the microcode or support code of some hardware system. The basic principles of both approaches are similar, but the former normally employs more powerful transformations, targeted at program optimizations, whereas the latter concentrates on translation from one instruction set to another, focusing on binary program portability.

Both the software and hardware approaches constitute virtual machines, on which the programs are executed, and both include some kind of translation and mapping of resources from the underlying system to the ones expected by the running programs. Smith and Nair (2005) describe in their book both virtual machines in software (called high-level virtual machines) and in hardware (low-level virtual machines). High-level virtual machines have the advantage that they can be ported more easily to very different underlying hardware architectures, but low-level virtual machines offer better performance, because they are implemented directly in hardware. A survey of adaptive optimization in virtual machines has been written by Arnold et al. (2005).

We will first discuss the various software systems because they directly relate to the work presented in this thesis, and then present hardware solutions for completing the discussion.

Dynamic Compilation in Software

One field in which dynamic code generation has been used since at least the 1960s is executable data structures. Instead of interpreting a data structure in order to control the steps of an algorithm, the data structure is translated to code which performs these steps. This has been done by Thompson (1968) for efficient string pattern matching with regular expressions. Another example is the translation of binary search in sorted arrays into nested conditional statements.

Brown (1976) introduced the name throw-away-compiling. He used it to describe his compilation technique, which allowed both compiled and interpreted code to coexist in a single program. This mixed code approach (Dakin and Poole, 1973; Dawson, 1973) was primarily intended to save memory by compiling only those parts of the programs which are most frequently used. In this early system, the programmer was responsible for annotating which parts of a program are to be compiled or interpreted. The compiled parts could be either pre-compiled before the program is run, or compiled at run-time, where one statement is compiled at a time and placed into a special memory region. When this region is filled, all compiled code is removed at once (thrown away), but Brown also suggests that more intelligent decisions should be made about which code is deleted and which is kept.

Software virtual machines operate on very different input languages. On the one extreme, in the case of binary translators, the input language is a machine language, so that the virtual machine has to do the same decoding as a processor. At the other end of the language spectrum, the machine translates a source language directly, as in Brown's approach mentioned above.

Binary translators work directly on machine language binaries. These are used especially for porting applications compiled for older architectures to new processor instruction sets. The Aries system (Zheng and Thompson, 2000), developed at Hewlett-Packard, translates PA-RISC code to IA-64 code and is used to execute native PA-RISC applications on IA-64 processors. Digital FX!32 (Chernoff and Hookway, 1997) is a software emulator for running IA-32 applications on the Alpha processor. The IA-32 Execution Layer (Baraz et al., 2003) is a software dynamic translator for running IA-32 software on Itanium-based (IA-64) systems.

Binary optimizers, where the input language is the same as the output language, are mostly used for the optimization of already compiled applications. The University of Queensland Binary Translation Project (UQBT) (Ung and Cifuentes, 2000, 2006) has developed a machine adaptable binary dynamic translation system. Other successful dynamic optimization systems are Dynamo (Bala et al., 2000) and its successor DynamoRIO (Bruening et al., 2003; Bruening, 2004), developed in cooperation between Hewlett-Packard Laboratories and the MIT Laboratory for Computer Science.

Dynamic Compilation in Hardware

Dynamic translation has been used in several hardware designs, mainly to maintain binary compatibility and to enable the use of new and more efficient hardware implementations. The idea is that the binary machine code, which has been compiled for some specific instruction set architecture (ISA) is transparently translated to another ISA prior to execution. Programs are held in memory in the old binary format and the translated code is stored in some fast cache memory, so that it can be reused when the code is executed several times. The granularity of translation is normally in size of some hardware-dependent measure, for example a memory page, in contrast to software-based dynamic compilation systems which generally translate one function, method or module at a time.

The DAISY system (Ebcioglu and Altman, 1996), implemented at IBM, translates machine code from other architectures to a VLIW architecture. The design supports different source languages, but the described implementation translates PowerPC code. Translation is performed by a Virtual Machine Monitor, which is a software that is stored in read-only memory.

The Transmeta Crusoe processors (Klaiber, 2000) use a dynamic compilation technique called code morphing, which translates IA-32 code to the machine language of a VLIW processor. The code morphing software is responsible for translating source instructions into target instructions (called *atoms*) and packing them as effectively as possible into long instruction words (called *molecules*). Atoms in one molecule are executed in parallel by different functional units of the processor.

The advantage of these automatic translation systems is that the processor implementor is free to change the underlying VLIW processor, as long as he adapts the translator software accordingly.

Hardware dynamic compilation has not yet been used in the particular context of functional programming, but of course the implementation of parallel graph reducers, which have been used for functional programs could be combined with dynamic code generation techniques.

Code Generation Techniques

Dynamic code generation techniques fall mainly into two groups: template-based techniqes and rule-based techniques.
In template-based approaches, code templates are created prior to execution. These templates contain holes into which values are inserted when on code generation time different templates are combined to construct the final machine code. This approach has been used, for example in DyC (Grant et al., 2000).
Rule-based techniques are closer to common code generation in static compilers. Starting with an abstract syntax tree of the program to be compiled, rules are applied in order to tile the syntax tree with instructions. These rules are either specified in some special declarative language for a code-generator generator (e.g. a bottom-up rewrite system (BURS)-based system, e.g. (Fraser et al., 1992)) or they are programmed ad-hoc in the compiler.

2.2.4 Dynamic Compilation and Functional Programming

There are fewer uses of dynamic compilation for functional languages than for object-oriented ones.
Wakeling (1998a) used dynamic compilation for the lazy functional language Haskell (Peyton Jones, 2003) in order to reduce the memory requirements of compiled code. In his system, Haskell source code is compiled to a compact intermediate format prior to execution and a dynamic compiler translates this code to machine code while the program is running. When the storage reserved for compiled code is exhausted, all compiled code is discarded and required code is re-generated (similar to Brown's throw-away compilation).
A remark by Spinellis (1990) hints at the possibility to use dynamic compilation for executing Haskell programs, but does not go into detail. Dockins and Guyer (2007) describe how to perform verification on the bytecode output of their Haskell compiler by adding type information to the bytecode. The Alice ML dialect is implemented on a virtual machine which also supports just-in-time compilation and distributed applications (Rossberg et al., 2006).
Burger (1997) has developed an infrastructure for profile-driven dynamic recompilation of the Scheme programming language (Burger and Dybvig, 1998). This work differs from the other work on functional programming languages because Scheme is not purely functional.
Dynamic loading and linking in functional languages has been considered for implementing functional shells (McDonald, 1983; van Weelden and Plasmeijer, 2004), and more general for dynamic applications which contain a small static core and load

all other code dynamically (Stewart and Chakravarty, 2005). One major advantage of using functional languages for such systems is their modularity, especially in languages which do not allow uncontrolled use of side effects. Rossberg (2006) describes a dynamic, type-safe linking approach for Alice ML.

One possible route for implementing dynamic functional programs is by compiling them for the Java Virtual Machine (JVM) (Wakeling, 1998b; Meehan and Joy, 1999). Unfortunately, the JVM architecture poses severe difficulties on the implementation of functional languages. Several features, which are key to the efficient implementation of functional languages (such as efficient memory allocation or cheap tail calls) are missing on this architecture (Schinz and Odersky, 2001; Shivers, 1996; Perry and Meijer). Therefore, we will target compilation to real machine code in this thesis.

Among the programming language abstractions which are targeted by our specialization techniques are type classes, which have mainly been implemented by passing additional parameters to overloaded functions (Hall et al., 1996; Augustsson, 1993), but where also specializing implementations exist (Jones, 1995; Meacham, 2007). Another promising language extension are open data types and open functions (Löh and Hinze, 2006), which allow the modular extension of algebraic data types and functions which operate on such extensible types. A similar development, but presented for object-oriented languages are extensible algebraic data types (Zenger and Odersky, 2001). Extensible programming with open case (Blume et al., 2006) solves similar problems, but with the addition of a case statement which is a first-class language element. The applicability of our techniques to these language features will be discussed as possible future work in Section 9.2.2.

The Fabius system (Leone and Lee, 1998) supports specialization, which is controlled through the use of curried functions. Earlier parameters to functions are considered to be more static than later arguments. Leone and Lee give as an example matrix multiplication using a curried function which takes two matrices. When applied to the first matrix, specialized code is created which is optimized for this input matrix. When the first matrix is sparse, execution speed is improved (Leone and Lee, 1994, 1996).

MetaML (Taha and Sheard, 1991; Taha, 1999) is a multi-stage programming language with explicit stage annotations. MetaML programs can explicitly construct programs and run them at a later time using language constructs for delaying and running computations. Currently, MetaML does not have a native implementation which could use its potential in a dynamic setting.

Dynamic Caml (Lomov and Moscal, 2002) is a library for Objective Caml that implements dynamic code generation for programs written in Objective Caml. The library provides a high-level interface which represents dynamically generated code in a form so that the run-time code generation process can be statically type-checked. It does not produce real machine code, but interprets the dynamically created Caml code.

Erlang (Armstrong et al., 1996) is a concurrent, dynamically typed, mostly functional programming language which was designed for programming telecommunication devices. One of its distinguishing features is support for code replacement, which means that it is possible to replace code modules without stopping the program. In order to both support code replacement and efficient execution, Erlang distinguishes calls to fully qualified functions (including a module and a function name) and unqualified functions (including only a function name). The former are dynamically looked up whereas references to the latter are resolved at compile time. Dynamically looked up calls always call the newest version of a compiled module, whereas statically resolved calls are never changed. It is therefore a recommendation to program long-running loops as tail calls to fully qualified modules. This is not enforced by the language definition or the compiler. For Erlang, a native code compiler called HiPE (High-performance Erlang) (Luna et al., 2005) exists, which also supports just-in-time compilation.

The Haskell package `hs-plugins` (Stewart, 2006) provides dynamic loading of Haskell modules. It also contains a *compilation manager* which can be used to compile Haskell source code into loadable modules prior to dynamically loading them. Since the compilation manager compiles the source code on a per-module basis, it is different from other dynamic code-generation libraries.

Leijen (2003) presents the implementation of a foreign function interface which allows Haskell functions to call functions written in other languages, and also calls from foreign functions back into Haskell code. The latter requires small amounts of run-time code generation for implementing closures which can be called like normal functions from foreign languages (call-back functions). The Glasgow Haskell compiler (GHC Developers, 2008) implements its foreign function interface in the same way. In both implementations, the use of dynamic code generation is hidden from the user.

Dubé (2002) presents an interesting framework for demand-driven type analysis. This is a version of type analysis for dynamically-typed functional languages, where type information is inferred during runtime based on estimates of how important it is at various program points. The idea is to avoid wasting time with costly analysis on program parts which are rarely or never used.

In all these previous works, several aspects of dynamic compilation and compilation techniques such as specialization and data type representations have been investigated. What is missing, is the integration of these ideas in a single framework. The goal of this thesis is to provide the foundations of such a framework.

2.2.5 Dynamic Compilation in Other Languages

Object-oriented languages were the first popular and widely used languages which supported dynamic features such as dynamic method dispatch that incur performance overheads when compared to languages such as Fortran, Pascal or C. Even though some of the older imperative languages supported a certain degree of dynamic procedure calls (through function pointers), most of the code was quite static. Newer object-oriented languages made the use of dynamic methods natural, so means were sought to improve their performance.

Smalltalk (Goldberg and Robson, 1983) is a class-based object-oriented programming language which uses an intermediate code. This code is generated from the source code by a compiler and in some implementations (Deutsch and Schiffman, 1984) it is translated further to machine code immediately before execution, eliminating the interpretation overhead of interpreter implementations. MultithreadSmalltalk (Ogata and Doi, 1994) also uses dynamic compilation.

Self is a dynamically typed object-oriented, prototype-based programming language (Ungar and Smith, 1991). The implementation (Chambers and Ungar, 1989) compiles Self source code to bytecodes which are translated to native machine code on demand and on a per-method basis. The machine code is held in a cache to be reused when a method is invoked in the future, and entries from the cache are discarded when the cache fills up.

The Java programming language (Gosling et al., 2000) was designed to support the distribution of compiled code and therefore includes the specification of a portable bytecode format, which is interpreted by the JVM (Lindholm and Yellin, 1999) in order to be executed. Most versions of the JVM include an option to translate the bytecode to native code, either on class loading (the dynamic loading method of Java) or on demand, when a certain method is to be executed which has not yet been compiled to machine code.

Various dynamically compiling Java virtual machines have been implemented: see for example Adl-Tabatabai et al. (1998), Adl-Tabatabai et al. (2003), Burke et al. (1999) and Agesen (1997). These implementations support a wide variety of optimizations, both conventional optimizations known from static compilation, and new ones which rely on the presence of a dynamic optimizer and/or instrumentation for profiling. Detlefs and Agesen (1999) describe inlining of virtual methods

Whaley (1999) has implemented an automatic runtime specialization algorithm for the use in Java virtual machines, which adapts Java programs to the actual run-time environment. Whaley's work is concerned with specialization of methods: either by inlining the most common method calls and thus specializing the method body to the context in which it was called, or by creating specialized versions of methods for commonly occurring parameters. Calls to these methods are then guarded by code which checks the preconditions, and if they fail, the more general method is called.

The .NET CLI (ECMA International, 2005) (Common Language Interface) specifies not only a bytecode format like Java (along with a human-readable notation) for .NET programs, but also explicitly specifies that bytecode modules are to be translated to machine code before execution. There is no information available on Microsoft's implementation of .NET, but several free implementations are available (DotGNU Project, 2004; Mono Project, 2007). They seem to perform roughly the same set of optimizations as the Java Virtual Machines mentioned above.

DyC (Grant et al., 2000) is a dynamic compilation system for the programming language C. DyC requires that the programmer annotates the source code so that the dynamic optimizer can efficiently specialize the source code for given fixed arguments. The Calpa system (Mock, 2002) uses value profiling to automatically generate these annotations, resulting in a fully automatic system.

Dynamo[1] (Leone and Dybvig, 1997) is a compiler architecture for dynamic program

[1]Not to be confused with the Dynamo system of Bala et al. (2000), described in Section 2.2.3.

optimization, based on the notion of staged compilation. Staged compilation divides the compilation process into several stages: the first stage corresponds basically to static compilation and performs classic program analysis and transformation techniques. Additionally, it collects information used in the second stage during run-time optimization. This setup allows to reduce the time and space requirements of all transformations happening at run-time while enabling beneficial optimization.

2.2.6 Formal Treatment of Dynamic Optimization

Even though the focus of this thesis is on practical matters, we give a short survey on the formal work on dynamic optimization.

There are two directions of research in this area. The first is mainly theoretic and builds on very abstract models of what computation means. The second aims at formalizing aspects of real dynamic compilation and optimization systems. While the former is very ambitious, it is very far away from practice, and the latter lacks theoretical sophistication, but is much more closer to what is implementable today.

In the first category, dynamic compilation has a close relationship to partial evaluation (Consel and Danvy, 1993), which dates back to the 1970s at least (see, for example, Ershov (1977), who called it a “partial computation principle”). Partial evaluation aims at separating static from dynamic evaluations, in order to perform static evaluation as early as possible, hopefully at compile time. Partial evaluation employs binding-time analysis to determine when all operands to an operation are available. The latest operand determines when evaluation can take place. Up to now, partial evaluation was used in static compilers, but its use in dynamic systems gives the possibility to perform certain evaluations at load-time or even later during run-time, when some execution parameters are fixed (for example, the machine architecture or the instruction set). Sperber and Thiemann (1997) use partial evaluation to improve the output of a Scheme compiler.

Balat and Danvy (1998) have used run-time code generation to produce bytecode for the Ocaml virtual machine. Their goal was to perform type-directed partial evaluation of ML programs.

The author has been developing a framework for generic dynamic optimization, which captures aspects such as profiling and incremental, adaptive dynamic compilation and optimization (Grabmüller, 2007). The work presented in this thesis is not directly connected to the earlier paper, and a connection of the theoretical and practical work is a topic for future work.

Modal-ML (Wickline et al., 1998) is a variant of the ML language which supports explicit code generation during run-time, allowing programmers to specify the stages of computation in a program. The different stages are reflected in the types assigned to program expressions.

In the more practical direction, formal treatment mostly involved the development of heuristics and cost models for tuning dynamic optimization.

Arnold et al. (2004) propose a model-driven policy for detecting recompilation opportunities which is used in the Jikes Research Virtual Machine (Burke et al., 1999;

Suganuma et al., 2000). They model both expected recompilation costs and the expected benefits of running optimized code, basing their heuristics on the compile times and profile data collected up to that point in execution. This seems to be the only published attempt to capture aspects of dynamic optimization systems formally.
Not directly related to dynamic compilation techniques, but possibly applicable to it are efforts to model the run-time behavior (such as time or space requirements) of programs. Using so-called cost semantics, Sands (1990) investigated the behavior of first-order and higher-order functional programs. Hope and Hutton (2005) derive step counting functions from function definitions. The techniques used in both approaches could be used to estimate the effects of optimizing functions.
In the TIL (typed intermediate language) project, type passing is used as an implementation technique (Morrisett, 1995; Tarditi, 1996; Tarditi et al., 1996). This implementation is based on the theory of intensional type analysis (Harper and Morrisett, 1995).
Gries and Gehani (1977) give an early discussion on type passing. They propose a limited form of polymorphic procedure declarations, where types in the parameter list can be marked so that they depend on the type at the call site. The body of a procedure can use the name of such a type, which essentially is a type parameter.

2.2.7 Applications

Modern programming language features such as reflection and aspect-oriented programming make high demands on the techniques used for implementation. The dynamic nature of such language features requires dynamic compilation in order to avoid excessive slowdowns. Ogel et al. (2005) present a system which dynamically weaves and compiles code in order to avoid the overheads of static monolithic aspect weaving.
Security is another important topic. When using networked machines, often connected over the Internet, it is important to make sure that programs received over the network have not been compromised. Several approaches, such as proof-carrying code or typed assembly language aim at supporting efficient verification of code before it is executed. Hornof and Jim (1999) examined how program certification and run-time code generation could be combined.
An interesting application of dynamic loading is dynamic software updating (Hicks, 2001; Stoyle et al., 2005; Bierman et al., 2003), which deals with the modification of running software systems. Dynamic software updating allows parts of a program to be replaced by corrected or extended code. Besides the modification and addition of code this involves the conversion of data in the running system when data type definitions are changed. This feature is crucial in systems which cannot tolerate any downtime. Hicks (2001) explores this application and describes how he developed and maintained a web server. He updated the web server's code several times without shutting it down. He did not use dynamic compilation techniques but worked with precompiled binaries instead.
Another application of dynamic compilation techniques is reduced power consump-

tion in embedded systems. Wu et al. (2005) use dynamic compilation for managing dynamic voltage and frequency scaling of the CPU, which results in significant reduction in power while maintaining high performance. Unnikrishnan et al. (2002) present a dynamic recompilation and linking framework which optimizes the energy behavior of a given application at run time.
A recurring pattern in all these applications is that they could be implemented using simple interpretation. The only reason for using dynamic compilation is to improve performance, even more so for dynamic optimization.

2.3 Analysis and Transformation

The process of compilation always consists of analysis and transformation. Analysis in this context is the calculation of program properties which are implicit in the program examined. Transformation is some meaning-preserving modification of the program which normally has the goal to improve it. Transformation normally makes use of the information gathered in the analysis phase in order to determine whether certain transformations are valid and will probably increase the program's efficiency. Some transformations are strictly local and can be based on the syntax of certain programming constructs, others require non-local or even global knowledge about the program. In the context of dynamic compilation, it is important to use very fast analysis and transformation algorithms, because their run time contributes to the total run time of the program under translation. This requires that sometimes algorithms must be used which are sub-optimal when compared to static compilers. Examples include linear register allocation (Poletto and Sarkar, 1999) in contrast to graph-coloring based algorithms (Chaitin, 1982) or first-order control flow analysis where a higher-order analysis could be used in static systems.
In summary, we can assume that static compilation has more resources at its disposal and is thus able to employ more complicated analysis and to use more aggressive transformations. On the other hand, a static scheme has less knowledge about the actual use of the program and must therefore either heuristically guess, or make limiting assumptions, such as closed-world compilation, where the complete source code of a program is available for analysis.
Dynamic compilers operate in more restricted environments, both time- and space-wise, but can make use of actual input values of programs and the actual machine configuration on which it runs.
We can see that dynamic compilation offers advantages in the following cases:

Selective elimination of abstractions Dynamic analyses and transformations do not need to take the complete source program into account. Compared to static specializers, dynamic specialization or cloning does not need the same amount of code space, because it can work selectively.

Exploiting dynamic information Since the dynamic compiler has access to all information dynamically available, such as input data or machine configuration, it can make better decisions when optimizing code.

Enabling dynamic language constructs Run-time compilation makes the whole system more dynamic, because it eases programming of distributed systems sharing programs by dynamic loading.

Among the disadvantages of the dynamic approach are the management overhead, resulting from code buffer management, compiler data structure management, and profiling. Additionally, it may be necessary to maintain information for selectively "undoing" certain optimization when assumptions on which these optimizations are based are invalidated – for example by loading new or updated code into the program. The main difficulty in implementing dynamic compilers is in the tradeoff between fast execution times of the translated program and compilation time. In this thesis we aim at providing some particular effective and efficient optimization techniques.

Program Analysis

The analysis of programs has a long history, dating back to the invention of the first higher-level programming languages in the 1950's and 1960's. Even the first commercial Fortran compiler performed a limited program analysis in order to optimize the compilation of indexing instructions in loops (Backus et al., 1957).
Program analysis consists of static (mostly compile-time) and dynamic analysis. The goal of static program analysis is to determine an approximation of the possible behavior of a program, without actually running it. Dynamic analysis tries to achieve more precise information by actually running a program for typical input data, gathering information for tailoring the compilation results to these data (*profiling* and *feedback-directed compilation*).
A detailed introduction to the field of program analysis is given by Nielson et al. (1999). Two important instances of program analysis are *data-flow analysis* and *control-flow analysis*. The first aims at determining the set of values which can possibly be held by the program variables during any program run, whereas the second tries to find out which functions, procedures or methods might be called in any call expression in the program. The two analyses are closely interrelated, because for example, the value of a boolean variable may control which alternative of a conditional expression is evaluated, thus resulting in different sets of functions to be called. In higher-order languages, where function parameters and local variables may be bound to functions and functions may be included in arbitrary data structures, the set of values contained in a variable determines the set of called functions when the variable is applied to an argument.
The advantage of static (or off-line) analysis is that it is allowed to spend more resources on the analysis, thus allowing better but more expensive algorithms to be used. On the other hand, the lack of actual input data restricts this kind of analysis because it must be overly conservative by considering any possible program run. Dynamic (or on-line) analysis is restricted to more efficient algorithms (which

normally have to be of linear time and space complexity), but has access to the input and profiling data of real program runs.
A combination of static and dynamic analysis has several advantages: it is possible to combine costly algorithms (run at compile time) with less costly, but also less powerful algorithms (run at run-time). The static analysis can remove overly general constructs from a program which are provably not needed for the specific application at hand, and the dynamic analysis can refine the compile-time optimizations by tailoring the program to the actual input data.

2.4 Typed Compilation

Typed compilation is an approach to programming language implementation which emphasizes the importance of strong typing at all levels of abstraction during the compilation process. The goal is to maintain accurate type information during all compilation phases. Types are transformed together with expressions when translating from one level of intermediate language to the next one. Finally, during code generation, type information can be used to support register allocation, data structure optimizations and tagless garbage collection (Tarditi et al., 1996; Tarditi, 1996; Morrisett, 1995) or type-based operations (Crary et al., 1998), such as polymorphic equality.
We will now summarize work in the area of typed compilation, and show the applications of types and type systems in compilers.

2.4.1 Use of Type Systems

Type systems have been used in a number of ways besides just checking the type-correctness of input programs. Hannan (1995) developed a type system for performing closure conversion. Hannan and Hicks (2000) give a type system for higher-order uncurrying and for arity raising (flattening argument tuples) (Hannan and Hicks, 1998). All these system use the type system to identify places for applying the respective transformation, where syntactic criteria do not suffice. The elimination of useless variables – for example, variables which are passed around recursive functions without contributing to the final result – using type systems has been investigated by Kobayashi (2001) and Fischbach and Hannan (2001).
Agat (1997) uses types for the register-assignment phase of a compiler. By annotating function types with the registers where input and output values are stored, the consistency of calling protocols can be ensured.

2.4.2 Continuation-passing Style

Many compilers for functional languages use some form of intermediate program representation which is based on the λ-calculus. These representations are often restricted versions, which require the programs to be in some normal form: A-normal form (ANF) (Flanagan et al., 1993), monadic normal form (Boquist, 1999) or

continuation-passing style (CPS) (Appel, 1992). Kennedy compares these languages for various properties (Kennedy, 2007).
Early work on CPS based program transformation can be found in Wand (1980).
In continuation-passing style, “every aspect of control flow and data flow [is made] explicit” (Appel, 1992, p. 2). By transforming the input program into a form where each function call is a tail-call, and each function is passed a function which performs the “rest of the computation” (the continuation function), the resulting program has made order of evaluation explicit and each intermediate result is named. Also, all call and return points are made explicit – calls to user-defined functions represent calls and calls to continuations represent returns. Most importantly, certain program transformations such as full β-reduction (function inlining), which are not generally sound in a call-by-value setting, are sound in CPS (Kennedy, 2007).
Our work is mainly inspired by the work of Appel (1992) and Kennedy (2007). Appel's algorithm is untyped, whereas Kennedy supports types, but does not handle polymorphic types.
Harper and Lillibridge (1993) describe a typed CPS conversion which supports polymorphic types.
Flanagan et al. (1993) show that CPS and A-normal form (ANF) are equivalent. In our view (and others, see Kennedy (2007)), CPS is better suited to optimization from a practical point of view, though.
Reppy (2001) uses CPS conversion in a limited way for optimizing loops in an otherwise direct-style compiler. He shows that one must not necessarily commit to one style or the other in one compiler.
Kelsey (1993) notes that CPS and static single-assignment (SSA) form as used in most modern optimizing compilers for imperative and object-oriented compilers are closely related and can be converted from one to the other quite easily. Appel (1998) has also investigated this topic.

2.4.3 Closure Conversion

General-purpose computers do not natively support higher-order functions, which access free variables (except for top-level bindings) and may be returned as results or stored in data structures. Therefore, one important part in the compilation of functional languages is the conversion of all functions to closed forms. This is normally performed by representing functions as heap-allocated records, with one reference to the code implementing the function, and several slots for storing the values of, or references to, its free variables.

Untyped closure conversion Shao and Appel (2000) and Appel (1992) describe untyped closure representations and conversion algorithms. We have extended the basic versions of these algorithms to our typed setting.

Typed closure conversion Minamide et al. (1996) describe a typed closure conversion algorithm. It uses existential and recursive types to type closures. We have

decided to stick to a simpler type system which is described in Section 4.4.2. As already mentioned above, Hannan (1995) uses types for closure conversion. Morrisett et al. (1999) describe the complete translation of terms in the second-order lambda-calculus (System F) to typed assembler language. The translation consists of several typed intermediate languages. Guillemette and Monnier (2007) present a type-preserving closure-conversion algorithm written in Haskell. Tolmach (1997) combines closure conversion with a type closure analysis algorithm. His goal is to represent closure-converted terms in the source language, so he uses algebraic data types to model closures and introduces the necessary creation of data records and selections.
The compiler described by Shao (1994) also uses a typed intermediate language and incorporates typed closure conversion. Morrisett (1995) has similar goals, but uses an intensional type analysis approach (Harper and Lillibridge, 1993). In this approach, the language supports a typecase construct, which allows the program to analyze the types of values and to make decisions based on types.

Run-time closure conversion Feeley and Lapalme (1992) use run-time compilation for the implementation of closures. A closure is represented as a short code fragment which pushes the values of free variables onto the stack and then invokes the code of the closure. On closure creation time, this code fragment is written to a freshly allocated region of memory.
Grabmüller (2006) has extended this approach so that the complete code of a function is generated at closure creation time, instead of generating a stub which calls a precompiled function.

General Other approaches use data and control flow analysis in order to optimize the creation and representation of closures. Cejtin et al. (2000) present a closure-conversion algorithm which makes use of data and control flow analyses in order to determine closure layout
Steckler and Wand (1997) introduces *lightweight* closure conversion, where several analyses are used to avoid redundant saving of variables to closures or passing unused closures to functions, thereby allowing multiple procedure-calling conventions in the same program. Siskind (1999) presents a similar closure conversion algorithm for a Scheme compiler in considerable detail.

2.4.4 Data Representation

Data representation and specializing representations for specific machine-supported data types have also been investigated. Type-directed unboxing (Leroy, 1997, 1992) and unboxed values as first-class citizens (Peyton Jones and Launchbury, 1991) have been proposed for efficiently implementing operations on native machine data types, for example in numerically intensive computations. When specializing all polymorphic functions, each value can be represented in its natural form, avoiding any overhead associated with repeated boxing and unboxing. A similar approach, called

record unboxing, has been proposed by Nguyen and Ohori (2007). Morrisett (1995) uses the type information passed at run time to polymorphic functions to perform type-dependent operations on different representations.
Morrison et al. (1991) have suggested to pass type information together with uniformly represented parameters, and to perform type-specific operations by inspecting the type information parameters. They also suggested the use of dynamic specialization on each call to a polymorphic function, or to optimize this by caching specializations. They state that "Neither of the above two implementation techniques is practical for reasons of space and time overhead, respectively" (Morrison et al., 1991, p. 349). We think that nowadays a typical personal computer or server has enough space and time at its disposal to make it practical.
Static specialization, where type-specific versions of functions are generated by the compiler at static compile time, have been successfully used in the Ada and C++ programming languages (Bray, 1983; Stroustrup, 1986). Ada supports generic modules, which are instantiated by importing them with concrete types. C++ supports templates, which are used to implement parametrized classes and functions.

Chapter 3

Dynamic Compilation of Functional Programs

This chapter proposes a design of a dynamic compilation system for functional programming languages. Based on this design, we have implemented a dynamically compiling virtual machine called Kafka[1]. The Kafka system executes programs written in a strict purely functional language (called *the Kafka language*) by incrementally compiling them to machine code. This implementation has been used for testing the design and for performing experiments in language design and execution performance. Even though we describe a specific system here, the general concepts described in this and later chapters are applicable to other systems as well.
We will first discuss the general architecture. Then we will have a look at the various phases of the compiler and the intermediate languages it uses. Details of the intermediate languages on which the compiler is based, the incremental compilation technique and the implementation are then discussed in detail in the following chapters.

3.1 Architecture

The architecture of the Kafka system is illustrated in Figure 3.1, which is a more detailed instance of the virtual machine schema described in Chapter 2. The Kafka system can read both source code written in its input language and compressed abstract syntax trees produced by a separate compiler.
The data space of the run-time compiler now has a distinguished component, called the specialization cache. This cache records for which types functions in the code buffer have been specialized, so that at most one compiled version exists for each instantiated function. In our prototype, this mapping between source function and specialized code is maintained only for polymorphic functions, but it could be used in general for all kinds of specialization.
The preprocessing applied to code loaded into the system depends on the kind of input. For source code the complete compiler front end (see the next section for details)

[1]For no particular reason.

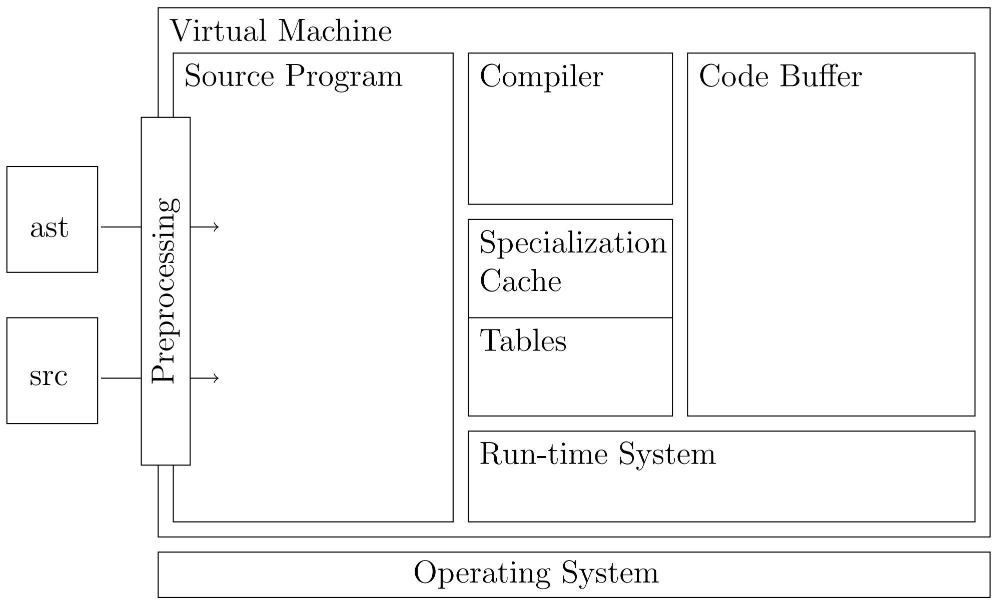

Figure 3.1: Dynamic specialization system architecture

is invoked, whereas for compressed abstract syntax input files only decompression and conversion to the internal representation is needed. The result of the preprocessing step is put into the memory area labelled *Source Program*. When the program is run, the run-time compiler takes the abstract syntax from the source buffer and generates machine code.
The code buffer holds machine code and is allocated as a fixed sized memory region. More sophisticated code buffer management is possible, but has not been implemented in the current version.
The run-time system provides services to the compiled program, such as garbage collection and an interface to operating-system services, for example input and output. Additionally, the run-time system writes profiling data to disk files when the program is terminating.
The tables are used for recording profiling information and statistics. Profiling can be selectively enabled and leads to the generation of instrumented machine code. Statistics are collected for several events: the number of code generations and garbage collections, the time spent in the code generator and the garbage collector as well as the actual running time of the code. Currently, profiling data is used for controlling incremental compilation. For more information on possible uses, the reader is referred to Section 9.2.5.

3.2 Compilation Process

The compilation process of our dynamically optimizing execution environment is illustrated in Figure 3.2.

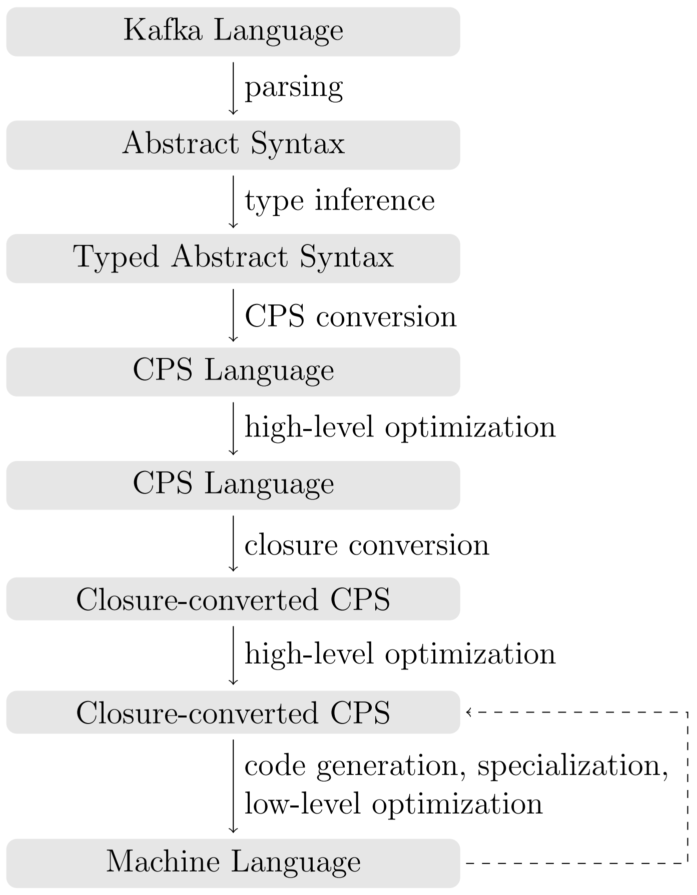

Figure 3.2: Compiler phases and intermediate languages

The Kafka language is semantically close to Standard ML (being a strict language), but has a syntax similar to Haskell. Kafka is a purely functional language which has no implicit side effects. Input and output are handled with continuation-passing functions and exceptions via an exception-handling mechanism similar to Standard ML.

The front end and the middle end of the Kafka system are conventional. A parser reads in the source code, checks its basic context-free syntax and converts it into an abstract syntax tree. This abstract syntax tree corresponds very closely to the source code, so that the type checker can generate helpful error messages.

The type checker implements a Damas-Milner type inferencing algorithm (Damas and Milner, 1982) for a let-polymorphic type system with modest extensions, such as overloaded arithmetic operators, type annotations and recursive algebraic data types. The output of the type checker is a type-annotated abstract syntax tree where some language constructs are desugared into more basic constructs. For example, equation-style function definitions are converted to let expressions which bind variables to λ-abstractions. The type checker also converts type abstractions and instantiations, which are implicit in the source code, into explicit type abstractions (λ-abstractions whose bound variables have the special type $\star$) and type applications (applications

to values of type $\star$).
In the rest of this thesis, the parser, the type checker and the untyped abstract syntax will not be discussed any further. They are quite conventional and are not affected by the dynamic code generation and optimization techniques applied in the other parts of the compiler.
The typed abstract syntax is fed into the continuation-passing style (CPS) transformation, which will be discussed in detail in Chapter 4. The resulting program in continuation-passing style is optimized and then converted into closure-passing style by the closure conversion. The closure-passing language (also discussed in Chapter 4) is a subset of the CPS language where functions and continuations are not allowed to have free variables. This representation is then again optimized.
The last step is the generation of machine code from the optimized, closure-converted representation. Since this representation is already fairly low-level, code generation is relatively easy and mainly consists of instruction selection, register and stack slot assignment and instruction generation.
Control can loop back from the machine language to the code generator when incremental compilation or run-time monomorphization occurs. In this case, closure-converted code (which is embedded in the machine code) is passed to machine code generation, and then control is passed to the newly generated code. This is indicated by the dashed arrow in the figure.
The dynamic compilation framework allows to delay code generation until a particular function or other part of the program is actually executed. This is used in two ways: functions (or parts of functions) can be compiled when first executed, thereby reducing the amount of code generation to those places where it is needed during an execution. Additionally, specialization of polymorphic functions is delayed until they are called with specific type parameters. Incremental compilation is discussed in Chapter 5, specialization of polymorphic functions in Chapter 6.
The compiler phases up to and including the optimization of closure-converted code are independent of our dynamic specialization technique. It can therefore be done ahead-of-time in a separate compiler, similar to the Java system, where the compiler produces bytecode off-line and the virtual machine generates machine code on-line. We have implemented this in our prototype system: a separate compiler can produce compressed abstract syntax files of closure-converted code which can be read into the virtual machine. This has the same effect as directly loading source code files, but avoids the overhead of parsing, type inference, CPS and closure conversion as well as high-level optimization.
More details on the implementation of our dynamically compiling architecture are given in Chapter 7.

Chapter 4

Typed Dynamic Continuation-passing Style

One goal of this work is to explore the possibilities of using continuation-passing style intermediate representations in the context of dynamic compilation. CPS is known to be well suited for data and control-flow analysis (Shivers, 1991), optimizations (Appel, 1992) and data representation optimizations (Shao, 1994; Shao and Appel, 1995).
The following question, which has not been addressed until now, is:

> How are CPS representations suited for performing dynamic optimizations, and especially, how can dynamic optimizations benefit from type-directed compilation in the setting of pure functional languages?

A novel compilation technique, called *typed dynamic continuation-passing style*, is described in this chapter. It offers the following features:

– Type-directed specialization of polymorphic functions and compile-time resolution of overloaded operators

– Type-directed data representation and layout optimizations.

When designing an intermediate compiler language, it is necessary to consider the level of abstraction that should be supported by the language. One possibility is to use a very high-level language, which is suited to analyses and transformations which rely on the high-level structure of the source program. At the other end of the spectrum, a low-level representation (such as quads or three-address instructions) allows machine-specific modifications of the program.
As an example, consider the combination of data representation optimization and the closure conversion phase of the compiler. The data representation optimization may decide (guided by type information) to put certain values into unboxed floating-point variables, which the code generator should eventually put into floating-point registers. Other values, unboxed integers or pointers will end up in general-purpose registers. During the closure-conversion phase, non-escaping functions are normally modified to

take their free variables as additional parameters, and all call sites of such functions are changed to pass these variables. Unfortunately, unless closure conversion knows about the representation of all variables, it is not possible to efficiently pass free variables to functions.

Therefore, we require that the uncompiled program fragments are stored in a fairly abstract form until they are compiled to machine code. It is one goal of this thesis to identify which transformations can be performed before type information is exploited (e.g., before type specialization and representation selection), and which transformations must be deferred until such decisions are made.

The rest of this chapter is organized as follows: we first introduce some notation and then present both the source and the target language of our typed continuation-passing transformation. The transformation between the explicitly typed source language and the CPS language requires both transformations on expressions and on types. Following that, we describe the closure conversion transformation which removes higher-order functions from the program. Finally we will sketch how machine-code can be generated from the closure-converted CPS code, and we describe the conversions performed on a small example program.

4.1 Notation

The following notational conventions are used in the following description of the source and CPS languages.

A superscripted type x^τ declares the x to be of type τ.

Overline notation is used for sequences. For example, $\overline{x}$ denotes a sequence of zero or more x entities.

4.2 Running Example

The transformations described in this chapter are numerous and complicated. In order to illustrate the effect of the transformations, we have prepared an example program which will be transformed step by step along with the descriptions and definitions of the transformation.

Figure 4.1 shows a complete program in the syntax of our prototype implementation. The program defines a list data type and a function for computing the sum of all elements of a list. The program also defines a two-element list of floating-point numbers and prints the sum of the list elements, calculated by the `sum` function, to the terminal. (The type annotation on list `l1` is necessary, because polymorphic values are not allowed.)

The fully typed program shown in Figure 4.2 is the result of type inference. This typed version of the language is the actual source language of the transformations described in this chapter, and standard type inference methods can be used to convert the input program from Figure 4.1 into the typed source program in Figure 4.2. The function `sum` is now a type function which expects the type of list elements and returns

```
data List a = Nil
            | Cons a (List a)
let
    sum l = case l of
              Nil -> 0
              Cons x xs -> x + sum xs
in
let val l1 = Cons 2.3 (Cons 1.0 Nil) :: List Prelude.Float
in Prelude.putf (sum l1) (\ v -> v)
```

Figure 4.1: Running example: input program

```
let fun sum:
        forall $15: *. ((List $15) -> $15) =
        (\ $15:
         * ->
         (\ l:
          (List $15) ->
          case l of
            Nil -> 0
            Cons x : $15 xs : (List $15) -> (((add $15) x)
                                             ((sum $15) xs)))) in
  let val l1: (List Float) = (Cons Float
                                   2.3
                                   (Cons Float 1.0 (Nil Float))) in
    ((putf ((sum Float) l1))
     (\ v: () -> v))
```

Figure 4.2: Running example: type-checked

a function for calculating the sum of lists of that type. Note that all applications of polymorphic functions and operators (`add` is the primitive polymorphic addition operator) are converted to explicit type applications (e.g., `sum Float` or `add $15`, where `$15` is a type variable).
The versions of the example program in the following sections are slightly edited for readability, but otherwise correspond directly to the transformation results of the compiler. Note that the examples use the concrete syntax of the compiler, whereas the formal definitions below use a more mathematical style, to improve readability.

4.3 Source Language

Our CPS transformation converts a typed language based on the λ-calculus into a continuation-passing language.
The source language of the transformation described here is not intended to be written by programmers, but is the output of some front-end, which annotates the original program text with type annotations. In our prototype implementation, the type inference mechanism transforms a Haskell-like language with optional type annotations into the language described here.
The source language provides polymorphic algebraic datatype definitions, polymorphic function definitions, value definitions, tuples and projections, value constructor applications and case expressions, λ abstractions, function application and numeric literals and primitive operations. Since the source language is a variant of the second-order λ calculus (also called System F (Barendregt, 1992)), it provides means for explicit type abstraction and application. Type and value abstractions or applications can be distinguished by the types of the formal or actual parameters, respectively. Type variables and type constants have type $\star$ (pronounced "type").
Type constants are the names of predefined types, for example integer, float, or user-defined types, which are declared with data definitions.

4.3.1 Syntax

The source language syntax is given in Figures 4.3 (types) and 4.4 (expressions). Source language types can be type variables, universally quantified types, the type of types ($\star$), tuples, type constructors applied to types and function types. Note that we require type constructors to be saturated: we do not support types of higher kinds. While in principle this should pose no problems, we have decided for a simpler system. Type constructors are derived from data type declarations and several type constructors are predefined: the unit type, booleans, exceptions, integer and floating-point types.
Source expressions can be variables, integer or floating-point constants, value constructor applications (which are required to be saturated), type or value function application, type or value abstraction, tupling, projections and the unit value. Integer literals are annotated with types, because they can be implicitly cast to floating-point

(value types)	Type $\ni \tau$	$::=$	α	type variable
		$\mid$	$\forall\alpha.\,\tau$	universal quantification
		$\mid$	$\star$	type of types
		$\mid$	$(\tau, \ldots, \tau)$	tuple type
		$\mid$	$TC\,\tau \ldots \tau$	applied type constructor
		$\mid$	$\tau \to \tau$	function type
(type constants)	TC	$::=$	unit	unit type
		$\mid$	bool	boolean type
		$\mid$	exn	exception type
		$\mid$	integer	integer type
		$\mid$	float	floating-point type
		$\mid$	T	user-defined data type

Figure 4.3: Source language of CPS transformation (types)

values (see Section 4.3.2 for details). A projection $\#i\#je$ selects the ith field from the j-tuple e. The more complex expressions are non-recursive value let bindings, recursive function let bindings, case expressions and the exception handling constructs raise (which raises an expression) and handle, which evaluates expression e_1 and returns its value if no exception was raised, and otherwise evaluates e_2 with variable x bound to the raised exception object and returns the resulting value.

Case alternatives consist of value constructors applied to variables and an expression on the right-hand side. When an alternative is matched against a value, the variables are bound to the respective fields of the constructed value and the right-hand side is evaluated with these bindings in scope.

Primitive operators are predefined constants, which are also mentioned in Figure 4.4. These appear simply as variables in the source program. Primitives are numeric arithmetic and comparison operators. Predefined constants also include the boolean constants false and true. More on primitive operators and numeric literals below in Section 4.3.2.

The source language is slightly unusual because most language constructs have type annotations. The reason is that the continuation-passing transformation discussed below needs to assign types to continuations and continuation parameters. Without explicit types in the source language, the continuation transformation would need to perform limited type inference. As we require a compiler front-end which produces the source language to do some kind of type checking or inference anyway, we simply require the semantic analysis to produce fully annotated programs. In the source syntax presented here, some type information is omitted when it can be inferred from the immediate context. This reduces notational clutter and makes programs much easier to read.

$$
\begin{array}{llcll}
\text{(expressions)} & \mathrm{Exp} \ni e & ::= & x & \text{variable} \\
 & & \mid & n^{\tau} & \text{integer literal} \\
 & & \mid & g & \text{floating-point literal} \\
 & & \mid & () & \text{unit value} \\
 & & \mid & (C\ \overline{e_i^{\tau_i}})^{\tau} & \text{constructor application} \\
 & & \mid & (e_1\ e_2)^{\tau} & \text{application} \\
 & & \mid & \lambda x^{\tau}.\, e^{\tau} & \text{abstraction} \\
 & & \mid & (e_1, \ldots, e_n)^{\tau} & \text{tuple} \\
 & & \mid & \#i\#j\ e^{\tau} & \text{projection} \\
 & & \mid & \mathsf{let}\ x^{\tau} = e_1\ \mathsf{in}\ e_2 & \text{let (non-recursive)} \\
 & & \mid & \mathsf{let}\ \overline{\mathit{Fun}}\ \mathsf{in}\ e_2 & \text{function definition (recursive)} \\
 & & \mid & (\mathsf{case}\ e^{\tau}\ \mathsf{of}\ \overline{\mathit{Alt}})^{\tau} & \text{case expression} \\
 & & \mid & \mathsf{raise}\ e & \text{raise exception} \\
 & & \mid & e_1^{\tau}\ \mathsf{handle}\ x \rightarrow e_2 & \text{handle exception}
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(alternative)} & \mathit{Alt} & ::= & C\ \overline{x^{\tau}} \rightarrow e^{\tau} \\
\text{(integer literals)} & n & ::= & 0 \mid 1 \mid \ldots \\
\text{(float literals)} & g & ::= & 0 \mid -1.1 \mid \ldots \\
\text{(function def.)} & \mathrm{FunDef} \ni \mathit{Fun} & ::= & f^{\tau} = e \\
\text{(datatype def.)} & \mathrm{DataDef} \ni \mathit{Data} & ::= & \mathsf{data}\ T\ \overline{\alpha} = \overline{C\ \overline{\tau}} \\
\text{(program)} & \mathrm{Prog} \ni P & ::= & \overline{\mathit{Data}}\ e
\end{array}
$$

$$
\begin{array}{rrcll}
\text{(constants)} & \mathsf{CS} & ::= & \mathsf{false} & \text{falsity} \\
 & & \mid & \mathsf{true} & \text{truth} \\
 & & \mid & \mathsf{add} & \text{addition} \\
 & & \mid & \mathsf{sub} & \text{subtraction} \\
 & & \mid & \mathsf{mul} & \text{multiplication} \\
 & & \mid & \mathsf{div} & \text{division} \\
 & & \mid & \mathsf{eq} & = \\
 & & \mid & \mathsf{neq} & \neq \\
 & & \mid & \mathsf{less} & < \\
 & & \mid & \mathsf{greater} & >
\end{array}
$$

Figure 4.4: Source language of CPS transformation (expressions)

4.3.2 Overloaded Numeric Literals and Primitive Operators

We want to exploit the opportunities which specialization gives for different numeric types, but we also try to keep the type system of the source and intermediate languages as simple as possible. Therefore, we have decided against implementing a full-blown type class system like that used in Haskell, and instead used a simpler alternative. Numeric integer literals are assigned the polymorphic type $\forall\alpha.\alpha$ by the type checker, so that they can be used both in integer and floating-point contexts. Similarly, primitive numeric operators have polymorphic types like $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$ or $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \mathsf{bool}$. Of course, these types are not precise enough, as the operations should only be allowed in numeric contexts: addition of two non-numeric types should not be allowed. Typing problems of this kind are found and reported during the specialization phase.

When specialization of polymorphic functions is performed, all type variables are replaced by concrete types before code is generated, so that the code generator has to deal with monomorphic code only. The code generator can then detect any wrongly-typed uses of numeric operations. The obvious drawback of this solution is that type errors are possibly found late in the compilation, at code generation time. Since code generation is interleaved with program execution, this may be really late. We emphasize that programs are type safe in the sense that no type incorrect operations are performed by the generated code.

In a production system, a type class or similar system should be used instead, but for our experiments, this solution proved to be sufficient. See Section 9.2.3.

One typing problem remains with this method: free type variables outside of any polymorphic functions are never replaced by types, because no specialization occurs on the top-level term. Therefore, before a program is evaluated, all free type variables of the program must be replaced by the type $\mathsf{integer}$. It must be the integer type because these free variables can only stand for numeric types[1], and because integer literals are the only constructs which are assigned type $\forall\alpha.\alpha$.

4.3.3 Static Semantics

The type system of the source language given in Figure 4.5 and Figure 4.6 is an adaptation of the type system for System F in Pierce (2002).

Term variable contexts Γ keep track of which type and term variables are in scope and give the types of term variables. Datatype contexts Δ record the type constructors of the predefined type constants and the user-defined data types. The type variables are included in a datatype context as well in order to check whether type constructor applications are saturated during the well-formedness check on types. Initially, Γ contains the bindings of all built-in constants (see Figure 4.7) and all data constructors from data type declarations with their declared types. Similarly, Δ contains all

[1]More precisely, an exception-raising program is assigned type α. This special treatment when the type of the complete program is polymorphic, is handled by our implementation properly.

$$\begin{array}{rrcll} \text{Ctxt} \ni & \Gamma & ::= & \emptyset & \text{empty context} \\ & & | & \Gamma, x\colon \tau & \text{term variable binding} \\ & & | & \Gamma, \alpha & \text{type variable binding} \end{array}$$

$$\begin{array}{rrcll} \text{DCtxt} \ni & \Delta & ::= & \emptyset & \text{empty context} \\ & & | & \Delta, T\,\alpha_1\,\ldots\,\alpha_n & \text{datatype binding} \\ & & | & \Delta, \alpha & \text{type variable binding} \end{array}$$

$\boxed{\Gamma; \Delta \vdash e\colon \tau}$

$$(\text{var})\frac{x\colon \tau \in \Gamma}{\Gamma; \Delta \vdash x\colon \tau} \qquad (\text{liti})\frac{\alpha \in \Delta}{\Gamma; \Delta \vdash n^{\alpha} : \alpha} \qquad (\text{litf})\frac{}{\Gamma; \Delta \vdash g : \mathsf{float}}$$

$$(\text{unit})\frac{}{\Gamma; \Delta \vdash () : \mathsf{unit}} \qquad (\text{capp})\frac{C\colon \tau_1 \to \cdots \to \tau_n \to \tau \in \Gamma \quad (\Gamma; \Delta \vdash e_i\colon \tau_i)_{i=1}^{n}}{\Gamma; \Delta \vdash (C\,\overline{e_i^{\tau_i}})^{\tau}\colon \tau}$$

$$(\text{tuple})\frac{(\Gamma; \Delta \vdash e_i\colon \tau_i)_{i=1}^{n}}{\Gamma; \Delta \vdash (e_1, \ldots e_n)^{(\tau_1,\ldots,\tau_n)}\colon (\tau_1, \ldots, \tau_n)} \qquad (\text{proj})\frac{\Gamma; \Delta \vdash e\colon (\tau_1, \ldots, \tau_j)}{\Gamma; \Delta \vdash \#i\#j\ e^{(\tau_1,\ldots,\tau_j)}\colon \tau_i}$$

$$(\text{abs})\frac{\Gamma, x\colon \tau_1; \Delta \vdash e\colon \tau_2}{\Gamma; \Delta \vdash \lambda x^{\tau_1}.e^{\tau_2}\colon \tau_1 \to \tau_2} \qquad (\text{app})\frac{\Gamma; \Delta \vdash e_1\colon \tau_1 \to \tau_2 \quad \Gamma; \Delta \vdash e_2\colon \tau_1}{\Gamma; \Delta \vdash (e_1\,e_2)^{\tau_2}\colon \tau_2}$$

$$(\text{tabs})\frac{\Gamma, \alpha; \Delta, \alpha \vdash e\colon \tau_2}{\Gamma; \Delta \vdash \lambda\alpha^{\star}.e^{\tau_2}\colon \forall\alpha.\tau_2} \qquad (\text{tapp})\frac{\Gamma; \Delta \vdash e_1\colon \forall\alpha.\tau_2 \quad \Delta \vdash e_2\colon \star}{\Gamma; \Delta \vdash (e_1\,e_2)^{\tau_2}\colon \tau_2}$$

$$(\text{let})\frac{\Gamma; \Delta \vdash e_1\colon \tau_1 \quad \Gamma, x\colon \tau_1; \Delta \vdash e_2\colon \tau_2}{\Gamma; \Delta \vdash \mathsf{let}\ x^{\tau_1} = e_1\ \mathsf{in}\ e_2\colon \tau_2}$$

$$(\text{letfun})\frac{\Gamma; \Delta \vdash \overline{Fun}\colon \Gamma' \ \Gamma \cup \Gamma'; \Delta \vdash e_2\colon \tau_2}{\Gamma; \Delta \vdash \mathsf{let}\ \overline{Fun}\ \mathsf{in}\ e_2\colon \tau_2}$$

$$(\text{case})\frac{\Gamma; \Delta \vdash e\colon \tau_s \quad \Gamma; \Delta \vdash \overline{Alt}\colon \tau}{\Gamma; \Delta \vdash \mathsf{case}\ e^{\tau_s}\ \mathsf{of}\ \overline{Alt}\colon \tau}$$

$$(\text{raise})\frac{\Gamma; \Delta \vdash e\colon \mathsf{exn}}{\Gamma; \Delta \vdash \mathsf{raise}\ e\colon \tau} \qquad (\text{handle})\frac{\Gamma; \Delta \vdash e_1\colon \tau \quad \Gamma, x\colon \mathsf{exn}; \Delta \vdash e_2\colon \tau}{\Gamma; \Delta \vdash e_1^{\tau}\ \mathsf{handle}\ x \to e_2\colon}$$

Figure 4.5: Source language type system (part 1)

$\boxed{\Gamma; \Delta \vdash \overline{\mathit{Fun}} \colon \Gamma'}$

$$\text{(funs)}\,\frac{\Gamma' = (f_i \colon \tau_i)_{i=1}^{n} \quad \Gamma \cup \Gamma'; \Delta \vdash (e_i \colon \tau_i)_{i=1}^{n}}{\Gamma; \Delta \vdash (f_i^{\tau_i} = e_i)_{i=1}^{n} \colon \Gamma'}$$

$\boxed{\Gamma; \Delta \vdash \overline{\mathit{Alt}} \colon \tau}$

$$\text{(alts)}\,\frac{(C_i \colon \tau_{i1} \to \cdots \to \tau_{in_i} \to \tau_s \in \Gamma \quad \Gamma, x_{i1} : \tau_{i1}, \ldots, x_{in_i} \colon \tau_{in_i} \colon e_i \vdash \tau)_{i=1}^{n}}{\Gamma; \Delta \vdash (C_i\, x_{i1}^{\tau_{i1}} \ \ldots \ x_{in_i}^{\tau_{in_i}} \to e_i^{\tau})_{i=1}^{n} \colon \tau}$$

$\boxed{\Delta \vdash \tau \colon \star}$

$$\text{(tcon)}\,\frac{T\,\alpha_1 \ \ldots, \ \alpha_n \in \Delta \quad (\Delta \vdash \tau_i \colon \star)_{i=1}^{n}}{\Delta \vdash T\,\tau_1 \ \ldots, \ \tau_n \colon \star} \qquad \text{(tvar)}\,\frac{\alpha \in \Delta}{\Delta \vdash \alpha \colon \star}$$

$$\text{(ttuple)}\,\frac{(\Delta \vdash \tau_i \colon \star)_{i=1}^{n}}{\Delta \vdash (\tau_1, \ldots, \tau_n) \colon \star} \qquad \text{(tforall)}\,\frac{\Delta, \alpha \vdash \tau \colon \star}{\Delta \vdash \forall \alpha . \tau \colon \star}$$

$$\text{(tfun)}\,\frac{\Delta \vdash \tau_1 \colon \ \star \quad \Delta \vdash \tau_2 \colon \star}{\Delta \vdash \tau_1 \to \tau_2 \colon \star}$$

Figure 4.6: Source language type system (part 2)

pre-defined type constructors (all constants of type $\star$ from Figure 4.7) as well as the user-defined data types with their type parameters.

The judgement $\Gamma; \Delta \vdash e \colon \tau$ specifies that in term variable context Γ and datatype context Δ, expression e has type τ. In order to keep the rules readable, we have not explicitly mentioned that all types appearing in the rules must be well-formed, as defined by the rules for judgement $\Delta \vdash \tau \colon \star$ in Figure 4.6.

The rules (var), (litf) and (unit) are conventional, rule (liti) assigns a polymorphic type to integer constants, as already mentioned in Section 4.3.2. Rule (capp) applies to constructor application, rules (abs) and (app) to value abstraction and application, respectively; similar for (tabs) and (tapp) for type abstraction and application. Note that for (tapp), the type parameter must be well-formed, which we explicitly check for. The let expressions are conventional, rule (letfun) uses the auxiliary judgement $\Gamma; \Delta \vdash \overline{\mathit{Fun}} \colon \Gamma'$ from Figure 4.6 which checks whether a sequence of mutually recursive functions is well-typed and returns the bindings of the defined functions to their types in Γ'. Case expressions are checked using rule (case), which again uses an auxiliary judgement $\Gamma; \Delta \vdash \overline{\mathit{Alt}} \colon \tau$ from Figure 4.6, which checks the typing of case alternatives. The exception raising statement in rule (raise) matches any type τ, because it aborts execution. Rule (handle) checks exception handling.

$$\begin{aligned}
\mathsf{add} &: \forall\alpha.\alpha \to \alpha \to \alpha \\
\mathsf{sub} &: \forall\alpha.\alpha \to \alpha \to \alpha \\
\mathsf{mul} &: \forall\alpha.\alpha \to \alpha \to \alpha \\
\mathsf{div} &: \forall\alpha.\alpha \to \alpha \to \alpha \\
\mathsf{eq} &: \forall\alpha.\alpha \to \alpha \to \mathsf{bool} \\
\mathsf{neq} &: \forall\alpha.\alpha \to \alpha \to \mathsf{bool} \\
\mathsf{less} &: \forall\alpha.\alpha \to \alpha \to \mathsf{bool} \\
\mathsf{greater} &: \forall\alpha.\alpha \to \alpha \to \mathsf{bool} \\
\mathsf{false} &: \mathsf{bool} \\
\mathsf{true} &: \mathsf{bool}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{unit} &: \star \\
\mathsf{exn} &: \star \\
\mathsf{integer} &: \star \\
\mathsf{float} &: \star \\
\mathsf{bool} &: \star
\end{aligned}$$

Figure 4.7: Types of built-in constants

4.3.4 Dynamic Semantics

The dynamic semantics of the source language are standard (see, for example Pierce (2002)). Note that when using a semantics which uses substitution for type applications, all type variables disappear from subterms before they are evaluated. Therefore, a primitive operation will never encounter a type variable as an argument, but only proper types. It is still possible, though, that a type error happens because of our handling of polymorphic numeric literals. See Section 4.3.2 above for details.

4.4 Continuation Language

We use an adapted and extended version of the continuation-passing language described by Kennedy (2007). In contrast to the CPS language used in Appel's book (Appel, 1992), this version is typed and represents exception handling by explicitly passing a failure continuation to each user function. The translation of the source language handle and raise constructs makes use of these failure continuations. In all other cases, the failure continuation is simply passed through to all called functions. In an implementation, the parameter which holds the failure continuation can be targeted to a particular register or memory location, so that passing it to called functions will avoid any copying overhead.
The syntax of the CPS language is given in Figures 4.8 (types) and 4.9 (terms). The language has been extended from Kennedy's with the following language constructs:

- n-ary tuples ($n > 1$) and projections
- parametric polymorphism
- support for the definition of polymorphic, recursive algebraic data types
- means for constructing and taking apart values of algebraic data types by case-expressions
- multi-argument continuations and functions (continuations also may take no arguments)

(value types)	CType $\ni \tau$	$::=$	α	type variable
		$\mid$	$\forall \alpha . \tau$	universal quantification
		$\mid$	$\bot$	bottom type
		$\mid$	$\star$	type of types
		$\mid$	$(\tau, \ldots, \tau)$	tuple type
		$\mid$	$T\ \tau \ldots \tau$	data type
		$\mid$	$\tau \rightarrow \tau$	function type
		$\mid$	$(\#\tau, \ldots, \tau\#)$	unboxed tuple type
(type constants)		$::=$	unit	unit type
		$\mid$	bool	boolean type
		$\mid$	exn	exception type
		$\mid$	integer	integer type
		$\mid$	float	floating-point type

Figure 4.8: CPS language (types)

- support for integer and floating-point literals
- primitive operations for performing integer and floating-point arithmetic and comparison

The types of the CPS language are an extension of source language types. The bottom type $\bot$ is used as the return type of functions and continuations in the CPS language, because they never return. Instead of returning to its caller, each function calls one of its continuations parameters or other functions. The main program is initially passed a special continuation which terminates the whole program.
Unboxed tuples are used for argument types of functions and are conceptional tuples. They model that functions can receive and return (as arguments to their continuation) multiple values, but unlike normal tuples, they cannot be stored into data structures. The idea of using (as well as the name and the syntax with $(\# \ldots \#)$ brackets of) unboxed tuples for this application is taken from Peyton Jones and Launchbury (1991).
The type of types $\star$ is used for type parameters. These need to be represented in the type system because polymorphic functions are represented as functions from types to functions.

4.4.1 Abbreviations

Types of CPS converted terms become very unwieldy because of explicit representation of success and failure continuations, and because unboxed tuples are used to represent multiple arguments. Therefore, the following abbreviations are defined for the sake of readability:

(variables)	$FVar$	$::=$	f	function name
	$CVar$	$::=$	k	continuation name
	$VVar$	$::=$	x	value name
(values)	Val	$::=$	$()$	unit value
		$\mid$	n^τ	integer literal
		$\mid$	g	floating-point literal
		$\mid$	$(x_1, \ldots, x_n)$	tuple value
		$\mid$	$C^\tau\, x_1 \ldots x_n$	constructor application
(terms)	Term $\ni K$	$::=$	$\mathsf{let}\ x^\tau = V\ \mathsf{in}\ K$	value binding
		$\mid$	$\mathsf{let}\ x^\tau = \pi_i\, x\ \mathsf{in}\ K$	projection
		$\mid$	$\mathsf{let}\ x^\tau = x_1 \oplus x_2\ \mathsf{in}\ K$	primitive operation
		$\mid$	$\mathsf{letcont}\ \overline{Cont}\ \mathsf{in}\ K$	continuation binding
		$\mid$	$\mathsf{letfun}\ \overline{Fun}\ \mathsf{in}\ K$	function binding
		$\mid$	$k\,\overline{x}$	continuation call
		$\mid$	$f\,\overline{x}$	function call
		$\mid$	$\mathsf{case}\ x\ \mathsf{of}\ \overline{Alt}$	case expression

(alternatives)	Alt	$::=$	$C \to k$
(primitives)	Prim $\ni \oplus$	$::=$	$+ \mid - \mid \times \mid /$
		$\mid$	$= \mid \neq \mid < \mid >$
(integer literals)	n	$::=$	$0 \mid 1 \mid \ldots$
(float literals)	g	$::=$	$0 \mid -1.1 \mid \ldots$
(function def.)	FDef $\ni Fun$	$::=$	$f^\tau \overline{x} = K$
(contin. def.)	CDef $\ni Cont$	$::=$	$k^\tau \overline{x} = K$
(datatype def.)	DDef $\ni Data$	$::=$	$\mathsf{data}\ T\,\overline{\alpha} = \overline{C\,\overline{\tau}}$
(program)	Prog $\ni P$	$::=$	$\overline{Data}\ K$

Figure 4.9: CPS language (expressions)

$\boxed{\Gamma \vdash V \ : \tau}$

$$\text{(val-unit)}\frac{}{\Gamma \vdash (): \mathsf{unit}} \qquad \text{(val-int)}\frac{}{\Gamma \vdash n^\tau : \tau} \qquad \text{(val-float)}\frac{}{\Gamma \vdash g : \mathsf{float}}$$

$$\text{(val-tuple)}\frac{(\Gamma \vdash x_i : \tau_i)\forall_{i=1}^{n}}{\Gamma \vdash (x_1, \ldots, x_n) : (\tau_1, \ldots, \tau_n)}$$

$$\text{(val-con)}\frac{(\Gamma \vdash x_i : \tau_i)\forall_{i=1}^{n} \quad C : \tau_1 \to \cdots \to \tau_n \to \tau \in \Gamma}{\Gamma \vdash C^\tau \ x_1 \ \ldots \ x_n : \tau}$$

Figure 4.10: CPS language type system (values)

$$\begin{array}{rcll}
\tau_1 \Rightarrow \tau_2 & \equiv & (\#\tau_1, (\#\tau_2\#) \to \bot, (\#\mathsf{exn}\#) \to \bot\#) \to \bot & \text{(function)} \\
\neg\tau & \equiv & (\#\tau\#) \to \bot & \text{(continuation)}
\end{array}$$

The first abbreviation stands for functions with exactly one parameter and one return value. All functions produced by the CPS transformation have this form, but later processing (such as optimizations like uncurrying or closure conversion) may later add arguments to functions and continuations. The second abbreviation uses the common notation of “not” for functions which do not return.

4.4.2 Static Semantics

The type system of the typed CPS language is shown in Figure 4.11. Rules of the form $\Gamma \vdash K$ ok mean that term K is well-typed in context Γ. The type system is also based on the rules in Figure 4.10, which specify the well-typedness of values. The type rules make use of the operator $\tau \leq \tau'$, which means that type τ is an instantiation of type τ'.

The typing rules for values are simple. The only uncommon feature, overloaded literals, require that the type of an integer literal is taken from its annotation. Again, we assume that all data constructor types are included in the environment Γ.

The value binding construct simply checks that the value is well-typed and that the type annotation on the bound variable is correct. The projection binding checks that the variable from which a value is projected is a tuple type of valid arity. Primitive operator bindings are checked for correct typing of arguments and result. Function and continuation applications require that the actual argument types match the function or continuation parameter types. For case expressions, the types of the mentioned continuations are checked against the fields of the corresponding constructor field types. The right-hand sides of function and continuation bindings are checked with the types of the defined functions/continuations in scope, to allow mutually recursive definitions.

$\boxed{\Gamma \vdash K \ \mathsf{ok}}$

$$\text{(letval)}\frac{\Gamma \vdash V : \tau \qquad \Gamma, x : \tau \vdash K \ \mathsf{ok}}{\Gamma \vdash \mathsf{let}\ x^{\tau} = V \ \mathsf{in}\ K \ \mathsf{ok}}$$

$$\text{(letproj)}\frac{x : (\tau_1, \ldots, \tau_n) \in \Gamma \qquad \Gamma, x : \tau_i \vdash K \ \mathsf{ok}}{\Gamma \vdash \mathsf{let}\ x^{\tau} = \pi_i\ x \ \mathsf{in}\ K \ \mathsf{ok}}$$

$$\text{(letprim)}\frac{\begin{array}{c} x_1 : \tau_1 \in \Gamma \qquad x_2 : \tau_2 \in \Gamma \\ \oplus : \tau^{\oplus} \in \Gamma \qquad \tau_1 \to \tau_2 \to \tau \leq \tau^{\oplus} \qquad \Gamma, x : \tau \vdash K \ \mathsf{ok} \end{array}}{\Gamma \vdash \mathsf{let}\ x^{\tau} = x_1 \oplus x_2 \ \mathsf{in}\ K \ \mathsf{ok}}$$

$$\text{(funapp)}\frac{f : \tau^f \in \Gamma \qquad (\#\tau_1, \ldots, \tau_n\#) \to \bot \leq \tau^f \qquad (\Gamma \vdash x_i : \tau_i)\forall_{i=1}^{n}}{\Gamma \vdash f\, x_1 \ \ldots \ x_n \ \mathsf{ok}}$$

$$\text{(contapp)}\frac{k : \tau^k \in \Gamma \qquad (\#\tau_1, \ldots, \tau_n\#) \to \bot \leq \tau^k \qquad (\Gamma \vdash x_i : \tau_i)\forall_{i=1}^{n}}{\Gamma \vdash k\, x_1 \ \ldots \ x_n \ \mathsf{ok}}$$

$$\text{(case)}\frac{\begin{array}{c} x : \tau \in \Gamma \\ (k_i : \tau^{k_i} \in \Gamma \qquad (\#\tau_{i1}, \ldots, \tau_{in_i}\#) \to \bot \leq \tau^{k_i})\forall_{i=1}^{n} \\ (C_i : \tau^{C_i} \in \Gamma \qquad \tau_{i1} \to \ldots \tau_{in_i} \to \tau \leq \tau^{C_i})\forall_{i=1}^{n} \end{array}}{\Gamma \vdash \mathsf{case}\ x \ \mathsf{of}\ C_1 \to k_1 \ldots C_n \to k_n \ \mathsf{ok}}$$

$$\text{(letfun)}\frac{\begin{array}{c} \Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash K \ \mathsf{ok} \\ (\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n, x_{i1} : \tau_{i1}, \ldots, x_{in_i} : \tau_{in_i} \vdash K_i \ \mathsf{ok} \\ \tau_i = (\#\tau_{i1}, \ldots, \tau_{in_i}\#) \to \bot)\forall_{i=1}^{n} \end{array}}{\Gamma \vdash \mathsf{let}\ f_1^{\tau_1}\ x_{11} \ldots x_{1n_1} = K_1 \ldots f_n^{\tau_n}\ x_{n1} \ldots x_{1n_n} = K_n \ \mathsf{in}\ K \ \mathsf{ok}}$$

$$\text{(letcont)}\frac{\begin{array}{c} \Gamma, k_1 : \tau_1, \ldots, k_n : \tau_n \vdash K \ \mathsf{ok} \\ (\Gamma, k_1 : \tau_1, \ldots, k_n : \tau_n, x_{i1} : \tau_{i1}, \ldots, x_{in_i} : \tau_{in_i} \vdash K_i \ \mathsf{ok} \\ \tau_i = (\#\tau_{i1}, \ldots, \tau_{in_i}\#) \to \bot)\forall_{i=1}^{n} \end{array}}{\Gamma \vdash \mathsf{let}\ k_1^{\tau_1}\ x_{11} \ldots x_{1n_1} = K_1 \ldots k_n^{\tau_n}\ x_{n1} \ldots x_{1n_n} = K_n \ \mathsf{in}\ K \ \mathsf{ok}}$$

Figure 4.11: CPS language type system (terms)

4.4.3 Dynamic Semantics

The dynamic semantics of the CPS language is defined by a standard, call-by-value evaluation strategy. The only extension to textbook semantics is that on application of type functions to type values, these type values must be substituted both at the term level and at the type level. Type variables may appear as term variables, for example in other type applications, and as type variables in type annotations.

4.5 CPS Transformation

The CPS transformation for translating source programs into CPS programs is shown in Figures 4.12 and 4.13. It is a variant of the higher-order one-pass call-by-value transformation of Danvy and Filinski (1992) which we have adapted from the presentation of Kennedy (2007, Fig. 8).

The transformation of expressions to CPS terms is performed mainly by two mutually recursive functions called $[\![\cdot]\!]$ and $(\!|\cdot|\!)$. Both are very similar, and translate an expression of the source language (first argument) into a term of the continuation language. The second argument is the name of the current exception continuation. The third argument is either a meta-level continuation which is called directly, or the name of an object-level continuation, for which an object-level continuation call has to be constructed. The reason for duplicating the transformation in two different contexts is that the algorithm avoids the generation of so-called *administrative reducible expressions (redexes)*, that is, statically reducible β redexes. Older transformations such as Appel (1992) rely on optimization transformations to eliminate these redexes. The transformation of Danvy and Filinski (1992) avoids their creation during construction of the CPS term and therefore does not need these optimizations (although other opportunities for optimizations remain, of course, depending on the input program).

In order to distinguish the meta and object levels, the following conventions are used (Kennedy, 2007):

- Bold lambdas $\boldsymbol{\lambda}$ are used to denote meta-level abstractions, normal lambdas λ for object-level abstractions.
- Meta-level applications are written with parenthesis $\kappa(x)$, object-level applications with juxtaposition $k\ x$.

There are three helper transformations: the first transforms source function definitions to CPS function definitions (Figure 4.15), the second transforms source types to CPS types (Figure 4.14) and the third transforms names of primitive operations to their operators (Figure 4.16). The transformation for functions simply invokes the $(\!|\cdot|\!)$ transformation on the function right-hand side, passing the two new continuation arguments of the transformed function. Conversion of source to CPS types only affects the types of functions, which are converted to multi-argument function types

$$
\begin{array}{rcl}
\llbracket\cdot\rrbracket & : & \mathrm{Exp} \to \mathrm{Var} \to (\mathrm{Var} \to \mathrm{Term}) \to \mathrm{Term} \\
\llbracket x \rrbracket\ h\ \kappa & = & \kappa(x) \\
\llbracket (\mathit{prim}\ \tau_1\ e_1\ e_2)^{\tau} \rrbracket\ h\ \kappa & = & \llbracket e_1 \rrbracket\ h \\
 & & \quad (\boldsymbol{\lambda} x_1.\ \llbracket e_2 \rrbracket\ h \\
 & & \qquad (\boldsymbol{\lambda} x_2.\ \mathsf{let}\ x^{\llbracket\tau\rrbracket} = x_1\ \llbracket \mathit{prim} \rrbracket\ x_2\ \mathsf{in}\ \kappa(x))) \\
\llbracket (e_1\ e_2)^{\tau} \rrbracket\ h\ \kappa & = & \llbracket e_1 \rrbracket\ h \\
 & & \quad (\boldsymbol{\lambda} x_1.\ \llbracket e_2 \rrbracket\ h \\
 & & \qquad (\boldsymbol{\lambda} x_2.\ \mathsf{letcont}\ k^{\neg\llbracket\tau\rrbracket}\ x = \kappa(x)\ \mathsf{in}\ x_1\ x_2\ k\ h)) \\
\llbracket \lambda x^{\tau_1}.\ e^{\tau_2} \rrbracket\ h\ \kappa & = & \mathsf{letfun}\ f^{\llbracket\tau_1\to\tau_2\rrbracket}\ x\ k\ h' = \llparenthesis e \rrparenthesis\ h'\ k\ \mathsf{in}\ \kappa(f) \\
\llbracket (e_1,\ldots,e_n)^{(\tau_1,\ldots,\tau_n)} \rrbracket\ h\ \kappa & = & \llbracket e_1,\ldots,e_n \rrbracket\ h \\
 & & \quad (\boldsymbol{\lambda} x_1,\ldots,x_n. \\
 & & \qquad \mathsf{let}\ x^{\llbracket(\tau_1,\ldots,\tau_n)\rrbracket} = (x_1,\ldots,x_n)\ \mathsf{in} \\
 & & \qquad\quad \kappa(x)) \\
\llbracket () \rrbracket\ h\ \kappa & = & \mathsf{let}\ x^{\mathsf{unit}} = ()\ \mathsf{in}\ \kappa(x) \\
\llbracket n^{\tau} \rrbracket\ h\ \kappa & = & \mathsf{let}\ x^{\llbracket\tau\rrbracket} = n\ \mathsf{in}\ \kappa(x) \\
\llbracket g \rrbracket\ h\ \kappa & = & \mathsf{let}\ x^{\mathsf{float}} = g\ \mathsf{in}\ \kappa(x) \\
\llbracket (C\ e_1 \ldots e_n)^{\tau} \rrbracket\ h\ \kappa & = & \llbracket e_1,\ldots,e_n \rrbracket\ h \\
 & & \quad (\boldsymbol{\lambda} x_1,\ldots,x_n.\ \mathsf{let}\ x^{\llbracket\tau\rrbracket} = C\ x_1\ \ldots\ x_n\ \mathsf{in}\ \kappa(x)) \\
\llbracket \#i\#j\ e^{(\tau_1,\ldots,\tau_j)} \rrbracket\ h\ \kappa & = & \llbracket e \rrbracket\ h\ (\boldsymbol{\lambda} z.\ \mathsf{let}\ x^{\llbracket\tau_i\rrbracket} = \pi_i\ z\ \mathsf{in}\ \kappa(x)) \\
\llbracket \mathsf{let}\ x^{\tau} = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \rrbracket\ h\ \kappa & = & \mathsf{letcont}\ j^{\neg\llbracket\tau\rrbracket}\ x = \ \llbracket e_2 \rrbracket\ h\ \kappa\ \mathsf{in}\ \llparenthesis e_1 \rrparenthesis\ h\ j \\
\llbracket \mathsf{let}\ \overline{d}\ \mathsf{in}\ e\ \mathsf{end} \rrbracket\ h\ \kappa & = & \mathsf{letfun}\ \llbracket \overline{d} \rrbracket\ \mathsf{in}\ \llbracket e \rrbracket\ h\ \kappa \\
\llbracket \mathsf{raise}\ e \rrbracket\ h\ \kappa & = & \llbracket e \rrbracket\ h\ (\boldsymbol{\lambda} z.\ h\ z) \\
\llbracket e_1^{\tau}\ \mathsf{handle}\ x \to e_2 \rrbracket\ h\ \kappa & = & \mathsf{letcont}\ j^{\neg\llbracket\tau\rrbracket}\ x = \kappa(x)\ \mathsf{in} \\
 & & \quad \mathsf{letcont}\ h'^{\neg\mathsf{exn}}\ x = \llparenthesis e_2 \rrparenthesis\ h\ j\ \mathsf{in}\ \llparenthesis e_1 \rrparenthesis\ h'\ j
\end{array}
$$

$$
\begin{array}{lcl}
\llbracket(\mathsf{case}\ e^{\tau_s}\ \mathsf{of} & & \\
\quad C_1\ \overline{x^{\tau_1}}_1 \to e_1 & & \\
\quad \ldots & & \\
\quad C_n\ \overline{x^{\tau_n}}_n \to e_n)^{\tau_r} \rrbracket\ h\ \kappa & = & \llbracket e \rrbracket\ h \\
 & & \quad (\boldsymbol{\lambda} z. \\
 & & \qquad \mathsf{letcont}\ j^{\neg\llbracket\tau_r\rrbracket}\ x = \kappa(x)\ \mathsf{in} \\
 & & \qquad \mathsf{letcont}\ k_1^{\neg(\#\overline{\llbracket\tau_1\rrbracket}\#)}\ \overline{x}_1 = \llparenthesis e_1 \rrparenthesis\ h\ j \\
 & & \qquad\qquad\quad \ldots \\
 & & \qquad\qquad\quad k_n^{\neg(\#\overline{\llbracket\tau_n\rrbracket}\#)}\ \overline{x}_n = \llparenthesis e_n \rrparenthesis\ h\ j\ \mathsf{in} \\
 & & \qquad\qquad\qquad \mathsf{case}\ z\ \mathsf{of}\ C_1 \to k_1 | \ldots | C_n \to k_n
\end{array}
$$

Figure 4.12: CPS transformation (part 1)

$$
\begin{array}{rcl}
\llparenthesis \cdot \rrparenthesis & : & \text{Exp} \to \text{Var} \to \text{Var} \to \text{Term} \\
\llparenthesis x \rrparenthesis\; h\; k & = & k\; x \\
\llparenthesis (\mathit{prim}\; \tau_1\; e_1\; e_2)^{\tau} \rrparenthesis\; h\; k & = & \llbracket e_1 \rrbracket\; h \\
 & & \quad (\boldsymbol{\lambda} x_1.\; \llbracket e_2 \rrbracket\; h \\
 & & \qquad (\boldsymbol{\lambda} x_2.\; \mathsf{let}\; x^{\llbracket \tau \rrbracket} = x_1\; \llbracket \mathit{prim} \rrbracket\; x_2\; \mathsf{in}\; k\; x)) \\
\llparenthesis (e_1\; e_2)^{\tau} \rrparenthesis\; h\; k & = & \llbracket e_1 \rrbracket\; h \\
 & & \quad (\boldsymbol{\lambda} x_1.\; \llbracket e_2 \rrbracket\; h \\
 & & \qquad (\boldsymbol{\lambda} x_2.\; x_1\; x_2\; k\; h)) \\
\llparenthesis \lambda x^{\tau_1}.\; e^{\tau_2} \rrparenthesis\; h\; k & = & \mathsf{letfun}\; f^{\llbracket \tau_1 \to \tau_2 \rrbracket}\; x\; k'\; h' = \llparenthesis e \rrparenthesis\; h'\; k'\; \mathsf{in}\; k\; f \\
\llparenthesis (e_1, \ldots, e_n)^{(\tau_1, \ldots, \tau_n)} \rrparenthesis\; h\; k & = & \llbracket e_1, \ldots, e_n \rrbracket\; h \\
 & & \quad (\boldsymbol{\lambda} x_1, \ldots, x_n. \\
 & & \qquad \mathsf{let}\; x^{\llbracket (\tau_1, \ldots, \tau_n) \rrbracket} = (x_1, \ldots, x_n)\; \mathsf{in}\; k\; x) \ldots) \\
\llparenthesis () \rrparenthesis\; h\; k & = & \mathsf{let}\; x^{\mathsf{unit}} = ()\; \mathsf{in}\; k\; x \\
\llparenthesis n^{\tau} \rrparenthesis\; h\; k & = & \mathsf{let}\; x^{\llbracket \tau \rrbracket} = n\; \mathsf{in}\; k\; x \\
\llparenthesis g \rrparenthesis\; h\; k & = & \mathsf{let}\; x^{\mathsf{float}} = g\; \mathsf{in}\; k\; x \\
\llparenthesis (C\; e_1 \ldots e_n)^{\tau} \rrparenthesis\; h\; k & = & \llbracket e_1, \ldots, e_n \rrbracket\; h \\
 & & \quad (\boldsymbol{\lambda} x_1, \ldots, x_n. \\
 & & \qquad \mathsf{let}\; x^{\llbracket \tau \rrbracket} = C\; x_1 \ldots x_n \mathsf{in}\; k\; x) \\
\llparenthesis \#i\#j\; e^{(\tau_1, \ldots, \tau_j)} \rrparenthesis\; h\; k & = & \llbracket e \rrbracket\; h\; (\boldsymbol{\lambda} z.\; \mathsf{let}\; x^{\llbracket \tau_i \rrbracket} = \pi_i\; z\; \mathsf{in}\; k\; x) \\
\llparenthesis \mathsf{let}\; x^{\tau} = e_1\; \mathsf{in}\; e_2\; \mathsf{end} \rrparenthesis\; h\; k & = & \mathsf{letcont}\; j^{\neg \llbracket \tau \rrbracket}\; x = \; \llparenthesis e_2 \rrparenthesis\; h\; k\; \mathsf{in}\; \llparenthesis e_1 \rrparenthesis\; h\; j \\
\llparenthesis \mathsf{let\; fun}\; \overline{d}\; \mathsf{in}\; e\; \mathsf{end} \rrparenthesis\; h\; k & = & \mathsf{letfun}\; \llbracket \overline{d} \rrbracket\; \mathsf{in}\; \llparenthesis e \rrparenthesis\; h\; k \\
\llparenthesis \mathsf{raise}\; e \rrparenthesis\; h\; k & = & \llbracket e \rrbracket\; h\; (\boldsymbol{\lambda} z.\; h\; z) \\
\llparenthesis e_1^{\tau}\; \mathsf{handle}\; x \Rightarrow e_2 \rrparenthesis\; h\; k & = & \mathsf{letcont}\; h'^{\neg \mathsf{exn}}\; x = \llparenthesis e_2 \rrparenthesis\; h\; k\; \mathsf{in}\; \llparenthesis e_1 \rrparenthesis\; h'\; k
\end{array}
$$

$$
\begin{array}{lcl}
\llbracket (\mathsf{case}\; e^{\tau_s}\; \mathsf{of} & & \\
\quad C_1\; \overline{x^{\tau_1}}_1 \to e_1 & & \\
\quad \ldots & & \\
\quad C_n\; \overline{x^{\tau_n}}_n \to e_n)^{\tau_r} \rrbracket\; h\; k & = & \llbracket e \rrbracket\; h \\
 & & \quad (\boldsymbol{\lambda} z. \\
 & & \qquad \mathsf{letcont}\; k_1^{\neg(\# \overline{\llbracket \tau_1 \rrbracket} \#)}\; \overline{x}_1 = \llparenthesis e_1 \rrparenthesis\; h\; k \\
 & & \qquad\qquad \ldots \\
 & & \qquad\qquad k_n^{\neg(\# \overline{\llbracket \tau_n \rrbracket} \#)}\; \overline{x}_n = \llparenthesis e_n \rrparenthesis\; h\; k\; \mathsf{in} \\
 & & \qquad\qquad\; \mathsf{case}\; z\; \mathsf{of}\; C_1 \to k_1 | \ldots | C_n \to k_n
\end{array}
$$

Figure 4.13: CPS transformation (part 2)

$$
\begin{array}{rcl}
[\![\cdot]\!] & : & \text{Type} \to \text{CType} \\
[\![\alpha]\!] & = & \alpha \\
[\![\star]\!] & = & \star \\
[\![(\tau_1, \ldots, \tau_n)]\!] & = & ([\![\tau_1]\!], \ldots, [\![\tau_n]\!]) \\
[\![T\ \tau_1 \ldots \tau_n]\!] & = & T\ [\![e_1]\!] \ldots [\![\tau_n]\!] \\
[\![(\tau_1 \to \tau_2)]\!] & = & (\#[\![\tau_1]\!], \neg[\![\tau_2]\!], \neg\mathsf{exn}\#) \to \bot
\end{array}
$$

Figure 4.14: CPS transformation of types

which receive the (transformed) argument type and the success and failure continuation as arguments. The success continuation takes the original return type as its argument.

We will now discuss in detail the transformation $[\![\cdot]\!]$ on source expressions. (The rules for $(\!|\cdot|\!)$ are analogous.) Variables are simply passed to the continuation of the transformations. Applications of primitive functions are transformed into the special binding for primitive operators, which applies an operator to two variables (the arguments) and binds the result to a variable. See Figure 4.16 for the translation of primitive operators. Applications are translated by translating both the function and the argument and by creating a return continuation to be passed to the called function. Abstractions are transformed to a function binding for one (newly named) function. Tuples use the natural extension of the $[\![\cdot]\!]$ transformation to lists of expressions, which lets us bind several variables at once. The arguments of the tuple constructor are then bound using the value binding construct. The translation of the unit value, floating-point literals and integer literals is straightforward, except for the fact that the type annotation for integer literals must be included in the target value binding.

Constructor applications are handled similarly to tuples. Projections are translated into projection bindings. Source let-bindings are translated by first translating the bound expression, which is passed into a fresh continuation which then evaluates the let body. Function bindings are translated using the auxiliary transformation in Figure 4.15. The raise and handle constructs are the only expressions which affect the failure continuations. Both constructs do not have a corresponding term in the target language; instead, the translation of raise simply invokes the failure continuation, whereas handle constructs a new continuation which is the translation of the handler clause and passes this continuation as the failure continuation to the translation of the first expression.

The definition of the non-tail-recursive transformation $(\!|\cdot|\!)$ is very similar to the tail-recursive one, except that some continuation creations can be avoided because the passed-in continuation is on the object level and can therefore be used directly in translated terms.

The treatment of user functions requires some more explanation, because these functions are possibly type functions as introduced by the type checker. It is necessary not to separate type parameters from each other. To see why, consider the function

$$
\begin{array}{rcl}
[\![\cdot]\!] & : & \text{FunDef} \to \text{FDef} \\
[\![f = \lambda\ x : \tau . e]\!] & = & f\ x\ k\ h = (\![e]\!)\ h\ k \\
 & & \quad \text{where } \tau \neq \star \\
[\![f = \lambda\ x_1 : \star .\ \ldots\ \lambda\ x_n : \star .\ \lambda\ x : \tau_x .\ e]\!] & = & f\ x_1 \ldots x_n\ k\ h = (\![\lambda\ x : \tau_x .\ e]\!)\ h\ k \\
 & & \quad \text{where } \tau_x \neq \star
\end{array}
$$

Figure 4.15: CPS transformation of functions

$$
\begin{array}{rcl}
[\![\cdot]\!] & : & \text{Var} \to \text{Prim} \\
[\![\mathsf{add}]\!] & = & + \\
[\![\mathsf{sub}]\!] & = & - \\
[\![\mathsf{mul}]\!] & = & \times \\
[\![\mathsf{div}]\!] & = & / \\
[\![\mathsf{eq}]\!] & = & = \\
[\![\mathsf{neq}]\!] & = & \neq \\
[\![\mathsf{less}]\!] & = & < \\
[\![\mathsf{greater}]\!] & = & >
\end{array}
$$

Figure 4.16: CPS transformation of primitive functions

map with the following type:

$$\forall a . \forall b . (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b$$

If this function were translated into curried type functions, the outer function applied to the first type a would have the type

$$\forall b . (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b$$

but it could not be compiled because the specialized version for type a does not know anything about the representation of type b. Therefore, our CPS transformation always translates type functions so that all type parameters (whose representation is fixed) are passed together, and separately from value parameters. This is reflected in the rules for the transformation of functions in Figure 4.15.

Running Example

The result of CPS conversion on the running example program is shown in Figure 4.17. To keep the example small, we have deleted type annotations on functions and continuations. Also, note that the alternatives of the case expression have expressions on the right-hand side of the arrow, not only continuation names. For readability, the continuations have been inlined in these cases, which can be enabled as an optimization in the CPS converter of our implementation (independently of the other optimizations described in Chapter 7).

```
letfun
  sum($15: *, k0: ~((List $15) => $15), h1: ~Exn) =
    letfun
      lam2(l: (List $15), k3: ~$15, h4: ~Exn) =
        case l of
          Nil ->
                letval lit7: $15 = 0 in
                k3 lit7
          Cons x xs ->
                 letval ty8: * = $15 in
                 letcont k9(x10: ((List $15) => $15)) =
                         letcont k11 (x12: $15) =
                                 let x13: $15 = x p+ x12 in
                                 k3 x13 in
                           x10 xs k11 h4 in
                   sum ty8 k9 h4
    in
      k0 lam2
in
  letcont k14 (l1: (List Float)) =
          letval ty15: * = Float in
          letcont k16 (x17: ((List Float) => Float)) =
                  letcont k18 (x19: Float) =
                          letfun
                            lam20 (v: (), k21: ~(), h22: ~Exn) =
                              k21 v
                          in
                            let x23: () = Prelude.putf (x19) in
                            lam20 x23 :.halt :.fail in
                    x17 l1 k18 :.fail in
            sum ty15 k16 :.fail in
    letval ty24: * = Float in
    letval lit25: Float = 2.3 in
    letval ty26: * = Float in
    letval lit27: Float = 1.0 in
    letval ty28: * = Float in
    letval Nil29: (List Float) = Nil ty28 in
    letval Cons30: (List Float) = Cons ty26
                                       lit27
                                       Nil29 in
    letval Cons31: (List Float) = Cons ty24
                                       lit25
                                       Cons30 in k14 Cons31
```

Figure 4.17: Running example: CPS converted

(value types)	CType $\ni \tau$	$::=$	$\dots$	
		$\mid$	$\{\tau, \tau, \dots\}$	closure type
		$\mid$	$\ulcorner \tau \urcorner$	closure-converted function
(terms)	Term $\ni K, L$	$::=$	$\dots$	
		$\mid$	letclos $\overline{Clos}$ in K	closure binding
(closure def.)	ClDef $\ni$ *Clos*	$::=$	$x^\tau = [x_1, \dots, x_n]$	
CC Context	*CCCtxt* $\ni \Gamma$	$::=$	$\emptyset$	empty context
		$\mid$	$\Gamma, x \colon \tau$	term variable binding

Figure 4.18: Extensions for closure-converted CPS code

4.6 Closure Conversion

The CPS language which results from the transformation in the previous section is already a low-level representation of the source program where evaluation order is made explicit, and where all intermediate values and control flow points are named. In order to compile the program to machine code, functions and continuations must be closed, by making accesses to free variables explicit. Therefore, we introduce closures for all defined functions, which contain the name of the function (a code pointer in the implementation) and the values of its free parameters.
This transformation is made by the closure conversion phase which is the subject of this section.
Figure 4.18 contains the extensions of the CPS language required for closure-converted code. Closure types are assigned to closures. The type contained in the closure type is the type of the function which is represented by that closure.
The second additional type is the type of closure-converted functions $\ulcorner \tau \urcorner$. The type τ must be a function type, and the markers over the type indicate that it is the type of functions of type τ, but in closure-converted form. That means that the functions expect an additional first argument, which is a closure for that function. Using this syntactic construct for marking closure-converted functions, we are able to avoid including recursive and/or existential types in our language, but still are able to type check closure-converted code.

4.6.1 Type checking Closure-converted Terms

The type system of closure-converted terms (Figure 4.19) is similar to the CPS type system shown above (Figure 4.11). New rules are required for the closure creation form letclos, for projection, for function and continuation application and for case expressions. All rules not mentioned here are the same as for the former type system. Type checking of closure-converted code is more difficult than the original CPS code,

$\boxed{\Gamma \vdash K \ \mathsf{ok}}$

$$\text{(letproj)}\ \frac{x\colon \{\tau_1, \ldots, \tau_n\} \in \Gamma \qquad \Gamma, x : \tau_i \vdash K \ \mathsf{ok}}{\Gamma \vdash \mathsf{let}\ x^{\tau} = \pi_i\ x\ \mathsf{in}\ K\ \mathsf{ok}}$$

$$\text{(funapp)}\ \frac{\begin{array}{c} f : \tau^f \in \Gamma \qquad \ulcorner(\#\tau_1, \ldots, \tau_n\#) \to \bot\urcorner \leq \tau^f \qquad (\Gamma \vdash x_i : \tau_i)\forall_{i=1}^{n} \\ f'\colon \tau^{f'} \in \Gamma \qquad \{\ulcorner(\#\tau_1, \ldots, \tau_n\#) \to \bot\urcorner\} \leq \tau^{f'} \end{array}}{\Gamma \vdash f\, f'\, x_1\ \ldots\ x_n\ \mathsf{ok}}$$

$$\text{(contapp)}\ \frac{\begin{array}{c} k : \tau^k \in \Gamma \qquad \ulcorner(\#\tau_1, \ldots, \tau_n\#) \to \bot\urcorner \leq \tau^k \qquad (\Gamma \vdash x_i : \tau_i)\forall_{i=1}^{n} \\ k'\colon \tau^{k'} \in \Gamma \qquad \{\ulcorner(\#\tau_1, \ldots, \tau_n\#) \to \bot\urcorner\} \leq \tau^{k'} \end{array}}{\Gamma \vdash k\, k'\, x_1\ \ldots\ x_n\ \mathsf{ok}}$$

$$\text{(case)}\ \frac{\begin{array}{c} x\colon \tau \in \Gamma \\ (k_i : \tau^{k_i} \in \Gamma \qquad \{\ulcorner(\#\tau_{i1}, \ldots, \tau_{in_i}\#) \to \bot\urcorner\} \leq \tau^{k_i})\forall_{i=1}^{n} \in \Gamma \\ (C_i : \tau^{C_i} \in \Gamma \qquad \tau_{i1} \to \ldots \tau_{in_i} \to \tau \leq \tau^{C_i})\forall_{i=1}^{n} \end{array}}{\Gamma \vdash \mathsf{case}\ x\ \mathsf{of}\ C_1 \to k_1 \ldots C_n \to k_n\ \mathsf{ok}}$$

$$\text{(letfun)}\ \frac{\begin{array}{c} \Gamma, f_1\colon \tau_1, \ldots, f_n\colon \tau_n \vdash K\ \mathsf{ok} \\ (\Gamma, f_1\colon \tau_1, \ldots, f_n\colon \tau_n, x_{i1}\colon \tau_{i1}, \ldots, x_{in_i}\colon \tau_{in_i} \vdash K_i\ \mathsf{ok} \\ \tau_i = (\#\tau_{i1}, \ldots, \tau_{in_i}\#) \to \tau\bot)\forall_{i=1}^{n} \end{array}}{\Gamma \vdash \mathsf{letfun}\ f_1^{\tau_1}\ x_{11} \ldots x_{1n_1} = K_1 \ldots f_n^{\tau_n}\ x_{n1} \ldots x_{1n_n} = K_n\ \mathsf{in}\ K\ \mathsf{ok}}$$

$$\text{(letcont)}\ \frac{\begin{array}{c} \Gamma, k_1\colon \tau_1, \ldots, k_n\colon \tau_n \vdash K\ \mathsf{ok} \\ (\Gamma, k_1\colon \tau_1, \ldots, k_n\colon \tau_n, x_{i1}\colon \tau_{i1}, \ldots, x_{in_i}\colon \tau_{in_i} \vdash K_i\ \mathsf{ok} \\ \tau_i = (\#\tau_{i1}, \ldots, \tau_{in_i}\#) \to \bot)\forall_{i=1}^{n} \end{array}}{\Gamma \vdash \mathsf{letcont}\ k_1^{\tau_1}\ x_{11} \ldots x_{1n_1} = K_1 \ldots k_n^{\tau_n}\ x_{n1} \ldots x_{1n_n} = K_n\ \mathsf{in}\ K\ \mathsf{ok}}$$

$$\text{(letclos)}\ \frac{\begin{array}{c} \Gamma, c_1\colon \tau_1, \ldots, c_n\colon \tau_n \vdash K\ \mathsf{ok} \\ ((x_{ij}\colon \tau_{ij} \in \Gamma)\forall_{j=1}^{k_i} \qquad \tau_i = \{\tau_{i1}, \ldots, \tau_{ik_i}\})\forall_{i=1}^{n} \end{array}}{\Gamma \vdash \mathsf{letclos}\ c_1^{\tau_1}\ = [x_{11}, \ldots, x_{1k_1}]\ \ldots\ c_n^{\tau_n}\ = [x_{n1}, \ldots, x_{nk_n}]\ \mathsf{in}\ K\ \mathsf{ok}}$$

Figure 4.19: Type system extensions for closure-converted terms

because functions of the same type, but with different free variables get assigned different types when closures are implemented as simple tuples. Therefore, we have introduced a special closure type, denoted by curly brackets: $\{\tau \rightarrow \tau'\}$ is the type of a closure for a function from τ to τ'. More elaborate type systems for closure conversion (Minamide et al., 1996; Hannan, 1995) have to rely on existential types to hide the types of free variables, or even have to add recursive types in order to express that a closure contains a pointer to a function which expects a closure of itself as an argument. We have side-stepped these issues in order to have a simpler type system. Closure types can have two forms: either only the function type of the function closed over is given between the curly brackets, or the function type and the type of all free variables are given. The first form is introduced at places where the types of the free variables are not known (for parameters of higher-order function, for example). At other occurrences, for example when a closure is created or when fields other than the first are projected from the closure, all types are given. This allows us to type check all uses of closures without the need for existential types.

A second problem is that closure-converted functions receive their closure as an additional parameter. This leads to recursive types, as the closure type again contains the type of the function. Using iso- or equi-recursive types, this can be reflected in the type system, but unfortunately this either complicates the type checker (which must unroll and roll types as required) or the intermediate language (which must contain constructs for unrolling and rolling types, to be inserted by the closure conversion phase). We have again decided to use a simpler type system: we introduce the special type $\ulcorner\tau_1 \rightarrow \tau_2\urcorner$, which represents a closure-converted function type. A closure of this type is then written $\{\ulcorner\tau_1 \rightarrow \tau_2\urcorner\}$. When type-checking applications of closure-converted functions to their parameters, the first parameter is interpreted specially: it must be a closure of the function type, and the other parameters must match the argument type of the function. Thus we can achieve type-safety without a more complicated type system and with an even simpler implementation for the type checker. The drawback of our representation is that closures always must be the first parameters of their respective closure-converted functions.

4.6.2 Closure Conversion Algorithm

The closure conversion code is shown in Figures 4.20 to 4.22. This conversion algorithm requires types, so the algorithm has an extra parameter, the variable context, which records the type of each bound variable while processing nested terms.

Most rules simply recurse on their nested term, binding variables to their annotated types. The rules for function and continuation bindings are converted to introduce closures, whereas function and continuation calls are modified so that they extract the code pointer from a closure before applying the respective function or closure.

On closure construction, all functions and continuations are renamed and closures with the original names are introduced. Each closure has its corresponding function pointer as the first element, and the free variables of the function form the remaining elements. In the function and continuation bodies, the free variables are then

$$
\begin{array}{rcl}
\{\!|\cdot|\!\} & : & \text{Term} \to \text{CCCtxt} \to \text{Term} \\
\{\!|\mathsf{let}\ x^\tau = V\ \mathsf{in}\ K|\!\}\ \Gamma & = & \mathsf{let}\ x^\tau = V\ \mathsf{in}\ \{\!|K|\!\}\ (\Gamma, x\colon \tau) \\
\{\!|\mathsf{let}\ x^\tau = \pi_i\ y\ \mathsf{in}\ K|\!\}\ \Gamma & = & \mathsf{let}\ x^\tau = \pi_i\ y\ \mathsf{in}\ \{\!|K|\!\}\ (\Gamma, x\colon \tau_i) \\
 & & \quad \text{where } y\colon (\tau_1, \ldots, \tau_n) \in \Gamma \\
\{\!|\mathsf{let}\ x^\tau = x_1 \oplus x_2\ \mathsf{in}\ K|\!\}\ \Gamma & = & \mathsf{let}\ x^\tau = x_1 \oplus x_2\ \mathsf{in}\ \{\!|K|\!\}\ (\Gamma, x\colon \tau) \\
\{\!|f\ x_1 \ldots x_n|\!\}\ \Gamma & = & \mathsf{let}\ g^\tau = \pi_1\ f\ \mathsf{in}\ g\ f\ x_1 \ldots x_n \\
 & & \quad \text{where } f\colon \{\tau\} \in \Gamma, g \text{ fresh} \\
\{\!|k\ x_1 \ldots x_n|\!\}\ \Gamma & = & \mathsf{let}\ l^\tau = \pi_1\ k\ \mathsf{in}\ l\ k\ x_1 \ldots x_n \\
 & & \quad \text{where } k\colon \{\tau\} \in \Gamma, l \text{ fresh}
\end{array}
$$

$$
\begin{array}{rcl}
\{\!|\mathsf{case}\ x\ \mathsf{of}\ C_1 \to k_1 \ldots C_n \to k_n|\!\}\ \Gamma & = & \mathsf{let}\ l_1 = \pi_1 k_1\ \ldots\ l_n = \pi_1 k_n\ \mathsf{in} \\
 & & \quad \mathsf{case}\ x\ \mathsf{of} \\
 & & \quad\quad C_1 \to l_1 \\
 & & \quad\quad \ldots \\
 & & \quad\quad C_n \to l_n
\end{array}
$$

Figure 4.20: Closure conversion (part 1)

extracted from the closure argument and bound to their original names, so that the rest of the body can refer to the names.

The function fvs, which is used for function and continuation bindings, returns a sequence of free variables of the argument. This function is applied to the bodies of the functions and continuations and the result (after removing the parameters with the $\setminus$ operator) are the variables to close over.

Similar to the continuation-passing transformation, closure conversion requires conversion of the types involved in the converted terms. The rules for converting types are shown in Figure 4.23. The types are simply converted recursively, except for the function type: converted function types are marked as closures over closure-converted functions. The $\{\!|\cdot|\!\}^{\to}$ transformation on types returns the closure-converted type of a function, without the closure braces.

The only operations which are allowed on closure types are the creation of closures (via letclos terms) and projections, where the projection of the first element is always allowed (it yields the function pointer) and projection of the other fields is only allowed in the body of closure-converted functions, where the closure type contains the types of the function and of its free variables.

The closure conversion algorithm is very simple and there exist many variants to this so-called basic algorithm. Refer to Chapter 2 for relevant literature.

4.6.3 Notes on Closure Conversion

There is a big difference between our closure conversion algorithm and the algorithms found in literature. Other approaches normally consist of two phases: the first introduces closures by modifying function definitions and applications and the creation of explicit closure structures (introduction phase). The second phase lifts all functions

$$
\begin{array}{l}
\{\!|\mathsf{letcont} \\
\quad k_1^{\tau_1}\ x_{11}\ldots x_{1k_1} = K_1 \\
\quad \ldots \\
\quad k_n^{\tau_n}\ x_{n1}\ldots x_{nk_n} = K_n \ \mathsf{in}\ K|\!\}\ \Gamma \quad = \\
\qquad\qquad \mathsf{letcont}\ l_1^{\{\!|\tau_1|\!\}^{\rightarrow}}\ k_1\ x_{11}\ldots x_{1k_1} = K_1' \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\quad l_n^{\{\!|\tau_n|\!\}^{\rightarrow}}\ k_n\ x_{n1}\ldots x_{nk_n} = K_n'\ \mathsf{in} \\
\qquad\qquad \mathsf{letclos}\ k_1^{\tau^{k_1}} = [l_1, y_{11}, \ldots, y_{1m_1}] \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\quad k_n^{\tau^{k_n}} = [l_n, y_{n1}, \ldots, y_{nm_1}] \\
\qquad\qquad\ \{\!|K|\!\}\ \Gamma_2 \\
\qquad\qquad \text{where } \Gamma_2 = \Gamma, k_1 \colon \{\!|\tau_1|\!\}, \ldots, k_n \colon \{\!|\tau_n|\!\}, \\
\qquad\qquad\qquad\qquad l_1 \colon \{\!|\tau_1|\!\}^{\rightarrow}, \ldots, l_n \colon \{\!|\tau_n|\!\}^{\rightarrow} \\
\qquad\qquad\qquad K_1' = \{\!|\mathsf{let}\ y_{11} = \pi_2 k_1\ \mathsf{in}\ldots \\
\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ y_{1m_1} = \pi_{m_1} k_1\ \mathsf{in}\ K_1'|\!\} \\
\qquad\qquad\qquad\qquad\qquad \Gamma_2, x_{11} \colon \{\!|\tau_{11}|\!\}, \ldots, x_{1k_1} \colon \{\!|\tau_{1k_1}|\!\} \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad K_n' = \{\!|\mathsf{let}\ y_{11} = \pi_2 k_n\ \mathsf{in}\ldots \\
\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ y_{nm_n} = \pi_{m_n+1} k_n\ \mathsf{in}\ K_n'|\!\} \\
\qquad\qquad\qquad\qquad\qquad \Gamma_2, x_{n1} \colon \{\!|\tau_{n1}|\!\}, \ldots, x_{nk_n} \colon \{\!|\tau_{nk_n}|\!\} \\
\qquad\qquad\qquad y_{11}, \ldots, y_{1m_1} = \mathrm{fvs}(K_1) \backslash \{x_{11}, \ldots, x_{1k_1}\} \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad y_{n1}, \ldots, y_{nm_n} = \mathrm{fvs}(K_n) \backslash \{x_{n1}, \ldots, x_{nk_n}\} \\
\qquad\qquad\qquad \tau^{k_1} = \{\{\!|\tau_1|\!\}^{\rightarrow}, \{\!|\Gamma(y_{11})|\!\}, \ldots, \{\!|\Gamma(y_{1m_1})|\!\}\} \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad \tau^{k_n} = \{\{\!|\tau_n|\!\}^{\rightarrow}, \{\!|\Gamma(y_{n1})|\!\}, \ldots, \{\!|\Gamma(y_{nm_1})|\!\}\}
\end{array}
$$

Figure 4.21: Closure conversion (part 2)

$$
\begin{aligned}
&\{\!|\mathsf{letfun}\\
&\quad f_1^{\tau_1}\ x_{11}\ldots x_{1k_1} = K_1\\
&\quad \ldots\\
&\quad f_n^{\tau_n}\ x_{n1}\ldots x_{nk_n} = K_n\ \mathsf{in}\ K|\!\}\ \Gamma \quad =\\
&\qquad\qquad \mathsf{letfun}\ g_1^{\{|\tau_1|\}^{\rightarrow}}\ f_1\ x_{11}\ldots x_{1k_1} = K_1'\\
&\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad g_n^{\{|\tau_n|\}^{\rightarrow}}\ f_n\ x_{n1}\ldots x_{nk_n} = K_n'\ \mathsf{in}\\
&\qquad\qquad \mathsf{letclos}\ f_1^{\tau^{f_1}} = [l_1, y_{11},\ldots,y_{1m_1}]\\
&\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad f_n^{\tau^{f_n}} = [l_n, y_{n1},\ldots,y_{nm_1}]\\
&\qquad\qquad \{\!|K|\!\}\ \Gamma_2\\
&\qquad\qquad \text{where } \Gamma_2 = \Gamma, f_1\colon \{\!|\tau_1|\!\},\ldots,f_n\colon \{\!|\tau_n|\!\},\\
&\qquad\qquad\qquad\qquad g_1\colon \{\!|\tau_1|\!\}^{\rightarrow},\ldots,g_n\colon \{\!|\tau_n|\!\}^{\rightarrow}\\
&\qquad\qquad\qquad K_1' = \{\!|\mathsf{let}\ y_{11} = \pi_2 f_1\ \mathsf{in}\ldots\\
&\qquad\qquad\qquad\qquad\quad \mathsf{let}\ y_{1m_1} = \pi_{m_1} f_1\ \mathsf{in}\ K_1'|\!\}\\
&\qquad\qquad\qquad\qquad\quad \Gamma_2, x_{11}\colon \{\!|\tau_{11}|\!\},\ldots,x_{1k_1}\colon \{\!|\tau_{1k_1}|\!\}\\
&\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad K_n' = \{\!|\mathsf{let}\ y_{11} = \pi_2 f_n\ \mathsf{in}\ldots\\
&\qquad\qquad\qquad\qquad\quad \mathsf{let}\ y_{nm_n} = \pi_{m_n+1} f_n\ \mathsf{in}\ K_n'|\!\}\\
&\qquad\qquad\qquad\qquad\quad \Gamma_2, x_{n1}\colon \{\!|\tau_{n1}|\!\},\ldots,x_{nk_n}\colon \{\!|\tau_{nk_n}|\!\}\\
&\qquad\qquad\qquad y_{11},\ldots,y_{1m_1} = \mathrm{fvs}(K_1)\backslash\{x_{11},\ldots,x_{1k_1}\}\\
&\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad y_{n1},\ldots,y_{nm_n} = \mathrm{fvs}(K_n)\backslash\{x_{n1},\ldots,x_{nk_n}\}\\
&\qquad\qquad\qquad \tau^{f_1} = \{\{\!|\tau_1|\!\}^{\rightarrow}, \{\!|\Gamma(y_{11})|\!\},\ldots,\{\!|\Gamma(y_{1m_1})|\!\}\}\\
&\qquad\qquad\qquad \ldots\\
&\qquad\qquad\qquad \tau^{f_n} = \{\{\!|\tau_n|\!\}^{\rightarrow}, \{\!|\Gamma(y_{n1})|\!\},\ldots,\{\!|\Gamma(y_{nm_1})|\!\}\}
\end{aligned}
$$

Figure 4.22: Closure conversion (part 3)

$$
\begin{array}{rcl}
\{\!|\cdot|\!\} & : & \mathrm{CType} \rightarrow \mathrm{CType}\\
\{\!|\alpha|\!\} & = & \alpha\\
\{\!|\bot|\!\} & = & \bot\\
\{\!|\star|\!\} & = & \star\\
\{\!|(\tau_1,\ldots,\tau_n)|\!\} & = & (\{\!|\tau_1|\!\},\ldots,\{\!|\tau_n|\!\})\\
\{\!|(\#\tau_1,\ldots,\tau_n\#)|\!\} & = & (\#\{\!|\tau_1|\!\},\ldots,\{\!|\tau_n|\!\}\#)\\
\{\!|T\ \tau_1\ldots\tau_n|\!\} & = & T\ \{\!|e_1|\!\}\ldots\{\!|\tau_n|\!\}\\
\{\!|\tau_1 \rightarrow \tau_2|\!\} & = & \{\{\!|\tau_1|\!\} \rightarrow \{\!|\tau_2|\!\}\}\\
\\
\{\!|\cdot|\!\}^{\rightarrow} & : & \mathrm{CType} \rightarrow \mathrm{CType}\\
\{\!|\tau|\!\}^{\rightarrow} & = & \tau' \quad \text{where } \{\tau'\} = \{\!|\tau|\!\}
\end{array}
$$

Figure 4.23: Closure conversion of types

(which are now closed and therefore independent of lexical scoping) to the top level (hoisting phase). We only perform the first phase, for the following reason: we *cannot* perform hoisting, because we do not close over free type variables. The reason is that type variables are not present in the code which is passed to the code generator, they are eliminated by the specialization algorithm described in Section 6. It is possible (and normal) for function and continuation definitions to have free type variables. The practical advantage when avoiding the hoisting phase, which brings all functions to the same lexical level, is that the scope of a type function (which is specialized en bloc) contains all internal function and continuation definitions, which need to be specialized anyway. So a single invocation of the specializer generates more code, thus reducing the overhead of the specializer as it needs to be invoked more rarely.

Running Example

Figure 4.24 shows the closure-converted running example. All functions are extended to receive their closures as the first arguments. Function applications are preceded by projections from closures, and closures are explicitly constructed using letclos terms. In order to save space, we only included the definition of the `sum` function and its closure.
When this resulting program is executed, the specialization mechanism for polymorphic functions presented in Chapter 6 will compile the summation loop into code which passes results in floating-point registers to continuations and uses floating-point instructions built into the processor. The list data type is also specialized to floating-point values, making efficient use of storage and aligning floating-point values so that fast access is possible. This monomorphization process of functions and polymorphic data types is discussed in detail in Chapter 6.

4.7 Generation of Machine Code

The generation of machine code from closure-converted CPS terms is straightforward, since the program is already in a low-level form. Many constructs of the CPS language correspond to just one machine-level concept. We will only briefly describe our code generator, because our focus is on the intermediate representation here.
The code generation function receives several parameters: a mapping from variable names to locations, a stack depth, a heap pointer (which records how many heap allocations have taken place since the start of a function) and the term to be translated. Locations are for example registers, stack slots, addresses relative to the heap pointer, absolute addresses or integer or floating-point literals. The mapping is called *compile-time environment*.
The various language constructs are translated as follows:

let expressions Value bindings either add a binding to the compile-time environment (for simple values), or are translated to the construction of a heap record (for tuples and constructed values). Bound variables are then represented as

```
letfun
  sum.8' (sum.0: {* ==> {(List $15) ==> $15}}, $15: *,
          k0.1: {~{(List $15) ==> $15}}, h1.2: {~Exn}) =
    letfun
      lam2.6' (lam2.3: {(List $15) ==> $15}, l.4: (List $15),
               k3.5: {~$15}, h4.6: {~Exn}) =
        let sum.0 = proj2,2 lam2.3 in
        case l.4 of
          Nil -> letval lit7.8 = 0 in
                  let k3.0' = proj1, k3.5 in k3.0' k3.5 lit7.8
          Cons x.9 xs.10 ->
                 letval ty8.12 = $15 in
                 letcont k9.4' (k9.13: {~{(List $15) ==> $15}},
                                x10.14: {(List $15) ==> $15}) =
                         let h4.6 = proj2,5 k9.13 in
                         let k3.5 = proj3,5 k9.13 in
                         let x.9 = proj4,5 k9.13 in
                         let xs.10 = proj5,5 k9.13 in
                         letcont k11.2' (k11.15: {~$15}, x12.16: $15) =
                                 let k3.5 = proj2,3 k11.15 in
                                 let x.9 = proj3,3 k11.15 in
                                 let x13.17 = x.9 p+ x12.16 in
                                 let k3.1' = proj1, k3.5 in
                                 k3.1' k3.5 x13.17 in
                           letclos k11.15 = [k11.2', k3.5, x.9] in
                             let x10.3' = proj1, x10.14 in
                             x10.3' x10.14 xs.10 k11.15 h4.6 in
                   letclos k9.13 = [k9.4', h4.6, k3.5, x.9, xs.10] in
                     let sum.5' = proj1, sum.0 in
                     sum.5' sum.0 ty8.12 k9.13 h4.6
    in letclos lam2.3 = [lam2.6', sum.0] in
        let k0.7' = proj1, k0.1 in k0.7' k0.1 lam2.3
in letclos sum.0 = [sum.8'] in ...
```

Figure 4.24: Running example: closure converted

pointers on the stack, and the stack slot binding is added to the compile-time environment.

projections Projections are translated to loads from memory.

function and continuation bindings The bodies of functions and continuations are translated and the addresses of the resulting code sequences are added to the compile-time environment.

function and continuation calls Calls are translated into a code sequence which loads the parameters into their correct argument positions, followed by a jump instruction.

case expressions These are translated into a load of the scrutinized variable and a sequence of tests, which determine the constructor of the value, followed by conditional tests.

closures, tuples and constructed values These are created on the heap.

primitive operations These are translated to their respective machine instructions, depending on their actual type (e.g., integer or floating-point addition).

Our extensive use of the compile-time environment allows us to load constants lazily, and even to delay operations such as projections: when a projection of a value, which is currently held in a register, is compiled, it can be translated to a register-indirect reference with constant offset. Using this technique, we make extensive use of the processor's addressing modes in order to avoid superfluous instructions and wasted registers.
Since arguments are passed in registers and stack slots, the algorithm for placing arguments in their correct locations must build an interference graph before emitting any move instructions. Otherwise, parameters which are needed later could be overwritten by earlier parameters.
We also use a special parameter assignment scheme to avoid move instructions on function/continuation calls. For example, continuation closures are passed to user functions in register `edx`. Continuations expect their closure argument in the same register, so on a continuation call, the closure pointer does not need to be moved at all. This optimization is possible, because continuation parameters can be recognized by their type. Other types of parameters with special argument locations are integers, floating-point registers or failure continuations. For integers, the first argument is passed in register `ecx` and others are passed on the stack, for floating-point values, six parameters are passed in registers `xmm2` to `xmm7` and the rest on the stack. Failure continuations are passed in memory, because they are infrequently accessed.

Running Example

We do not present the complete machine code for the running example, because it is very long. Instead, we have picked out a few small fragments of machine code and

relate it to the closure-converted version of the program in Figure 4.24.
The function `sum.0`, which is a type function in the closure-converted program, is translated to the following code which calls the polymorphic specializer. Details of this process are described in Chapter 6.

```
sum.0:
      pushad                        ; save registers
      push   dword ptr [esp+32]     ; push type parameter
      push   esp                    ; push additional parameters
      push   1H                     ; ...
      push   1aH
      push   1bH
      push   0f0f0f0f0H
      push   11H
      push   1H
      call   [0b7b7101cH]           ; call code generator
      add    esp,20H
      popad                         ; restore variables
      mov    eax,dword ptr [esi+24] ; fetch address of
      jmpn   eax                    ;   compiled code and jump
```

When the specializer has created the floating-point version of the `sum` function, the core of the summation loop looks as follows:

```
k11.24:
      movsd  xmm7,dqword ptr [edx+8] ; fetch operand from closure
      addsd  xmm7,xmm2               ; add parameter
      mov    eax,dword ptr [edx+16]  ; load continuation
      push   dword ptr [eax+4]       ;   from closure
      mov    edx,dword ptr [edx+16]  ; move parameters
      movsd  xmm2,xmm7               ;   to correct locations
      mov    eax,dword ptr [esp]
      add    esp,4H
      jmpn   eax                     ; jump to continuation
```

We can see that the compiler has created machine code with floating-point instructions, because the sum function has been used in a floating-point context.

Chapter 5

Incremental Compilation

One property of dynamic compilers is that it is not necessary to compile the complete program on each run. Only the parts of the program which are reached during a particular program run need to be translated. For large programs with a small working set (that is, a small set of functions which are evaluated on a "typical" run), this feature can save a lot of time, since no time is spent on generating code for unused functions. Additionally, code buffer space is saved.

The idea is to delay the generation of code until it is certain that the code will be executed—this is similar to lazy evaluation, where computation is delayed until a result is required, for example for output. This process of delayed code generation is called *incremental compilation* (Johnston, 1979).

In this chapter we describe the necessary extension to the intermediate CPS language from Chapter 4, we show how code is generated for delayed expressions, and the heuristics used to decide which expressions to delay selectively.

5.1 Language Extension

We have realized incremental compilation by adding a special syntactic construct to the intermediate language: delay expressions. A delay expression wraps a CPS term. The extension to the CPS language is modest and is shown in Figure 5.1. The syntax is extended with delay expressions, and the extension to the type system is also shown.

The dynamic semantics would not be changed when represented as a big-step semantics, because the meaning of delay expressions is the same as the meaning of their subexpressions. Delay expressions only have an effect on code generation, so a low-level operational semantics which realizes the dynamic translation would have to model the delayed generation of target code.

Syntax extension:

$$\begin{array}{llcll} \text{(terms)} & \text{Term} \ni K, L & ::= & \ldots & \\ & & \mid & \mathsf{delay}\ K & \text{delay expression} \end{array}$$

Type system extension:

$$\text{(delay)}\frac{\Gamma; \Delta \vdash K\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{delay}\ K\ \mathsf{ok}}$$

Figure 5.1: CPS language extension for incremental compilation

5.2 Code Generation for Delay Expressions

When the code generator encounters a delay expression, it does not generate the code for the wrapped term, but instead generates a call to the run-time code generator. In order to access the free variables of the delayed expression, the call is preceded by a sequence of instructions which pushes the values of all free variables to the stack and by extending the compile-time environment to record these new mappings for the variables. The parameters for the call to the code generator are:

– The number of words on the run-time stack.
– The current compile-time environment, which records the mapping from variable names to locations.
– The term which is wrapped in the delay expression.
– A pointer to the current state of the run-time compiler.
– A machine code label for the first parameter-pushing instruction in this sequence (needed for patching later).

After this code sequence, instructions are generated for loading the result value of the call into a register and for jumping to the returned address.
The instruction sequence generated for the delay expression is quite short (about a dozen instructions), so code generation is finished quickly.
When later during the execution of the program the call to the code generator is reached, control is transferred to the run-time compiler. It proceeds by (a) placing new delay expressions in the body term (if any, see below), (b) generating code for that body, (c) patching the calling instruction stream with a jump to the freshly generated code, and (d) returning the address to the calling code.
The following pseudo-machine code demonstrates the compilation of delay expressions. Suppose we have a term

$$t = \mathsf{delay}\ (k\ x)$$

which represents a delayed continuation call. The following code will be generated:

```
    push <k>                    ; push the values of free variables
    push <x>
L0:                             ; address to patch later
    push <stack-depth>
    push <compile-time-env>
    push <term>
    push <execution-state>
    push L0
    call <generator>            ; initiate code generation
    pop-arguments               ; remove all parameters from the stack
    jmp <return-value>          ; invoke newly-generated code
```

When the code is executed, the free variables k and x will be pushed to the stack and the code generator is called with its arguments. It will then generate code similar to the following (ignoring closure pointers for clarity), which loads x from its stack location to its correct parameter location and k into a register and then jumping to the continuation through the `eax` register.

```
L1:
   mov eax, [esp+4]             ; load continuation address
   mov ebx, [esp]               ; move parameter to correct location
   mov [esp+4], ebx
   add esp,4                    ; adjust stack pointer
   jmp eax                      ; invoke continuation
```

After this code is generated, we need to update the original instruction stream to avoid another code generation for the delayed expression when it is executed again. So the code sequence for the delay expression above will be modified in memory to read:

```
    push <k>                    ; old code
    push <x>
L0:
    jmp L1                      ; <--- new instruction
    <garbage>                   ; old code, partly overwritten
    push <term>
    push <compile-time-env>
    push <execution-state>
    push L0
    call <generator>
    pop-arguments
    jmp <return-value>
```

The instructions after the `jmp L1` are now unreachable and could be re-used by a sophisticated code buffer manager. Note that writing the jump instruction into memory will probably partly overwrite old instructions, which are no longer well-formed instructions. Since they will never be executed, this is no problem, though. On the second execution of the delayed expression, the only inefficiency left is the pushing of the free variables and the patched jump instruction.

5.3 Placing of Delay Expressions

Of course, wrapping a delay expression around every subterm of a program is inefficient, because the call of the code generator and code generation take some time. We need to selectively add delay expressions to subterms, and this decision can be based on measurements, static heuristics, or both.
In our implementation we have experimented with the following heuristics: delay expressions are only wrapped around bodies of user functions, because these tend to be the places where code generation for large expressions can be avoided by inserting a single delay expression. Other expressions, which represent straight-line code (value bindings, primitive operations, projections, etc.) do not gain from being delayed, because code can be generated quickly for these. The other language construct where we add delay expressions is in the branches of case expressions, because some branches of case expressions will never be executed on a successful program run, for example tests for error conditions and their handling. Other branches will only seldomly be executed, so we hope to save time for the majority of program runs. The drawback of this scheme is that some branches will be executed on all program runs, so delaying them wastes time.
Therefore, in addition to static heuristic measures, we have added some profiling support. For all functions, continuations and branches of case expressions, the system can be instructed to generate instrumentation code. The instrumentation instructions count how many times during a program run a particular function, continuation or case alternative is executed. When the program halts, these counters are written to disk, and on the next run, the counters are read in again and made available to the code generator. The algorithm for placing delay expressions can now consult the counters to decide whether to delay the body of a particular function, continuation or branch, or not. In the simplest case, only code which has not been executed in the last run will be delayed. More static or dynamic measures could of course be taken into account, but we have not yet investigated this topic any further.
Additionally, we can apply the simple rule that bodies of polymorphic functions are never delayed, since their code is generated on demand when they are applied anyway (see Chapter 6 on Run-time Monomorphization). Code generation for polymorphic functions is thus always incremental, so delaying them would not save on code generation time or code size, but would instead incur overheads because two calls to the code generator would be made in sequence.
Incremental compilation is initiated by wrapping a delay expression around the complete program. Since code generation for delay expressions triggers additional placement of delay expressions on subterms, the complete program will eventually be

selectively delayed.

5.4 Discussion

Note the close relationship of the implementation of delay expressions to lazy evaluation and closure representations. Similar to lazy evaluation, work is deferred until it is needed. In our case, code generation is avoided, in lazy evaluation computation is avoided. When the code is eventually needed, we generate it and update the instruction stream to jump to the newly created code. This is similar to updates of heap nodes in implementations of lazy evaluation, where indirection nodes are used in some implementations (for example the Spineless tagless G-Machine, (Jones, 1992)). Similar to closure representation, the code generated for a delay expression closes the abstract syntax tree of the wrapped expression with respect to its free variables (by pushing them on the stack and recording their locations) and the compile-time environment (which is stored in the instruction stream and used later for code generation).
In order to gather some experimental results on the effectiveness of delayed compilation, we have performed some experiments which are presented in Chapter 8.

Chapter 6

Run-time Monomorphization

Based on the intermediate program representation defined in Chapter 4, we now develop a dynamic specialization strategy which eliminates polymorphism from functional programs at run-time. Specialization of polymorphic programs with respect to types is called monomorphization, because it results in monomorphic programs (Boquist, 1999; MLton Developers, 2006).

Monomorphization is a simple but effective method for reducing the run-time cost of language abstractions. In functional languages, abstractions such as functions, polymorphism, type classes and parameterized modules are used very frequently, and their efficient implementation is the key to the usability of such languages.

Except for the treatment of type classes in Section 6.1.3, the techniques in this thesis should be applicable to any polymorphic functional languages with algebraic data types and higher-order functions.

The main goal of our dynamic compilation system is the reduction of unnecessary indirections due to abstractions. Polymorphic data types and functions require either a uniform representation of data and functions, or specialized versions of all used functions for the data types actually occurring in a program. In an open system, such specialization at static compile time is not possible, because the compiler cannot know which data types may be used in any possible program run. Therefore, in our approach, code and data specialization is deferred until run-time, where all needed data types are known and monomorphic code for all polymorphic functions can be produced.

The basic idea is that polymorphic functions are translated to code generators, and that calls to polymorphic functions pass types for which the target function should be specialized. Each call to a polymorphic function therefore generates new machine code, specialized to the given types. Caching techniques are used to avoid too many specialized variants of a polymorphic function.

Decompilation and deoptimization techniques (Kotzmann and Mössenböck, 2005) can be used once a violation of the assumptions of the specializer is detected and the needed invariants can be reestablished by recompiling the code.

Our specialization implements several language features, which are all useful in their own right, but can be combined for best effect: run-time monomorphization creates

monomorphic instances of polymorphic functions for all usages. Monomorphization can also be used for specializing polymorphic data types. Dynamic type class resolution is an implementation technique for type classes which is different from the dictionary-based implementations normally used (Hall et al., 1996; Augustsson, 1993). This kind of type class resolution can be compared to the work of Jones (1995), but since we do monomorphization of (parametric) polymorphic functions anyway, discovering method invocations and the identification of overloaded functions is much simpler and can be done in parallel to the monomorphization.
In earlier work, all these abstractions required different implementation techniques. In the following sections, we will show that our approach subsumes them in a fairly simple specialization model.
In our prototype implementation, we have implemented specialization of polymorphic functions, resolution of overloaded literals and arithmetic operations and specialization of polymorphic data types, including data representation and layout optimizations.
Other possible applications of dynamic specialization can be found in Section 9.2.2.

6.1 Specialization of Polymorphic Functions

Polymorphism in functional languages means one of two language features: parametric polymorphism, which allows the definition of functions over unknown types, and ad-hoc polymorphism or overloading, which refers to the possibility of giving the same name to different entities. The run-time specialization approach in this section can eliminate all overhead which may result from both of these abstraction constructs.

6.1.1 Polymorphic Functions

Polymorphic functions are functions defined over values of unknown types. Consider for example the classic *map* function, which applies a function to all elements of a list. This function has the type:

$$map : \forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow \mathsf{List}\ \alpha \rightarrow \mathsf{List}\ \beta$$

The universally quantified type variables α and β are arbitrary, but fixed in the type. Polymorphism is a very powerful abstraction mechanism. It allows the programmer to write very general functions, which can be partly oblivious to the types of the values they operate on. The disadvantage is that it is difficult to generate a machine code sequence which realizes the algorithm of such a function. Since the type of the function does not have any information on the machine representation of the parameters, an implementation must compromise: either a uniform representation is chosen for all data types, or separate versions of the function must be created, each for a different type. In the former case, the representation of values cannot be exploited by using special instructions or calling conventions which may be provided

by the machine for particular data types, in the latter, many nearly identical versions of code must be generated.
Our first use of specialization is used to remove parametric polymorphism from the program code actually executed. The idea is simple: when a polymorphic function is instantiated (that is, applied to a type parameter), the actual type parameter is substituted for the formal parameter in the body of the function, and this body is then passed to the code generator. Since the code generation is based on the specialized instances of the function definitions, the code generator can take advantage of the fact that all type information is present. It can therefore use the most efficient machine representations and calling conventions for a given data type for the machine executing the program. For example, integer values can be passed in different registers than floating-point values, and different machine instructions can be used for the addition, comparison, etc. of machine representations. These specializations are already done in many static whole-program compilers, but in dynamic compilers, the approach is much more flexible.

The Specialization Mechanism

The specialization mechanism is invoked when a type function is called. This can only take place after some code has already been executed, because programs are required to be monomorphic at the top level, and only embedded function definitions can be polymorphic. This is the mechanism for function monomorphization:

1. All functions which receive any type parameters are compiled specially by the code generator. They are always functions which return functions (this is guaranteed by the CPS conversion), so their body always consists of local function definitions, closure creations for the defined function and a call to the success continuation which passes one of the closures. Instead of directly generating code for the function body, a call to the run-time specializer is generated. This call sequence consists of pushing the arguments for the specializer onto the stack, calling the code generator which generates specialized code and then jumping to the freshly generated code. The parameters for the code generator include: all actual type parameters, the current compile time environment (mapping variable names to locations), a pointer to the AST of the function itself, the current execution state of the virtual machine, and the current stack depth. The compile time environment and the stack depth are required so that the code generation for the specialized function body can access the parameters. This is important, because the function body will call its success continuation.

2. The code generator takes the list of actual type parameters from the run-time stack and the list of formal type parameters from the function AST, and then substitutes the former for the latter in the AST. The resulting AST is monomorphic,[1] and the code generator is called on it, returning the start address of the

[1]Actually, the resulting code may contain polymorphic functions, but before any of them could be executed, it would be monomorphized recursively.

```
let id x = x
in id id ()
```

Figure 6.1: Identity function example

newly generated code block. This address is passed back to the code generated in step 1, which jumps to it.

The specializer avoids the generation of duplicate code by maintaining a mapping from function definition/type list-pairs to addresses of compiled code. When it is applied to a concrete list of types and a function definition to specialize, it looks up the pair in the mapping and returns the address of already generated code if possible; otherwise, the code is generated and its address is stored in the mapping.
Another optimization, described in more detail in Section 7.2, is *inline caching*. Depending on a command line option, the code generator generates a series of comparisons before the call to the specializer. These comparisons compare the actual type parameters against values stored in the machine code, which are written to the instruction stream after each specialization. In the common case where a function is repeatedly called with the same type parameters, these checks succeed and jump around the specializer call, loading a constant address to jump to instead, which also has been placed by the specializer. This sequence of checks is much faster than a call to the specializer. For similar reasons, inline caching has been used for the implementation of object-oriented languages (Detlefs and Agesen, 1999).

Specialization Example

Consider the small example program in Figure 6.1. The polymorphic identity function `id` is defined, and it is first applied to itself, and then the result is applied to the value `()` (which denotes the single element of the unit type).
Figure 6.2 lists the code which is produced for this example by the code generator. The register `edi` is used as the heap pointer, registers `edx` and `ecx` are used for passing parameters.
Lines 1–8 of the machine code create closures for the identity function and a continuation, which is not of interest here. The continuation definitions have been omitted. Heap overflow checks and calls to the garbage collector have also been deleted to save space, and the code was compiled with inline caching switched off. The first few labels refer to continuation code which will be ignored here, but note the code starting at label `id.0` (line 22). The push instructions and the call invoke the specializer, and after it returns, the address of the generated code is loaded into register `eax` and then jumped to.
The code which is generated on the first call to the identity function in the example program `(id id)` results in the code shown in Figure 6.3. The lines up to the first label construct a closure for the monomorphized identity function which starts at label `lam2.4` (line 8). This closure is then passed to the success continuation (lines 4–7).

```
            add    edi,8H                     ; allocate heap memory
            mov    dword ptr [edi-4],0b7a0215cH; initialize closure
            mov    dword ptr [edi-8],13H
            add    edi,10H
            lea    ebx,[edi-24]               ; allocate heap memory
            mov    dword ptr [edi-8],ebx      ; initialize closure
            mov    dword ptr [edi-12],0b7a02088H
            mov    dword ptr [edi-16],15H
            sub    esp,4H                     ; set up arguments
            mov    ebx,dword ptr [esi+12]
            mov    dword ptr [esi+16],ebx
            lea    edx,[edi-16]               ; load success continuation
            mov    dword ptr [esp],14H
            lea    ecx,[edi-24]
            jmp    [0b7a0215cH]               ; jump to function
      k6.14: [b7a02088]
            ...
      k9.19: [b7a020ec]
            ...
      k11.23: [b7a02144]
            ...
      id.0: [b7a0215c]
            pushad                            ; save registers
            push   0b7a0215fH                 ; push arguments
            push   dword ptr [esp+36]
            push   esp
            push   1H
            push   19H
            push   1aH
            push   0b7a0216aH
            push   12H
            push   1H
            call   [0b7c6701cH]               ; call specializer
            add    esp,24H
            popad                             ; restore registers
            mov    eax,dword ptr [esi+24]     ; load address of generated
            jmpn   eax                        ;   code and jump
```

Figure 6.2: Assembler code for identity example

```
            add    edi,8H
            mov    dword ptr [edi-4],0b7a0226bH
            mov    dword ptr [edi-8],13H
            lea    ecx,[edi-8]
            mov    eax,dword ptr [edx+4]
            add    esp,4H
            jmpn   eax
      lam2.4: [b7a0226b]                    ; code for identity function
            mov    ecx,dword ptr [esp]      ; load argument for
            mov    eax,dword ptr [edx+4]    ;   success continuation
            add    esp,4H
            jmpn   eax                      ; jump to continuation
```

Figure 6.3: Assembler code for identity example: integer version

```
            add    edi,8H
            mov    dword ptr [edi-4],0b7a021c6H
            mov    dword ptr [edi-8],13H
            lea    ecx,[edi-8]
            mov    eax,dword ptr [edx+4]
            add    esp,4H
            jmpn   eax
      lam2.1': [b7a021c6]                ; floating-point version
            mov    eax,dword ptr [edx+4] ; no argument moving necessary
            jmpn   eax
```

Figure 6.4: Assembler code for identity example: floating-point version

In our example, the second invocation of the identity function specializes the function for the unit type, which results in exactly the same code because the unit type and closures are both represented as 32-bit machine words.

Figure 6.4 shows the code which would result if the identity function was specialized for the type `float`. The code for building the closure and returning remains unchanged, because it is independent of the type parameter in this case. The monomorphized identity function (which starts at label `lam2.1'` in the example) is different from the integer version: it does not even need to copy its input, because the first floating point parameter for both functions and continuations is always passed in the same register.

We can see in these examples how all generated code is independent of the types insofar as no tag handling or type passing is required for monomorphized code.

```
 1  data Tree a = Leaf a
 2              | Node (Tree (a, a)) (Tree (a, a))
 3  let size t :: Tree a -> Integer =
 4                case t of
 5                  Leaf x -> 1
 6                  Node l r -> size l * 2 + size r * 2
 7  in
 8    let val t1 = Node (Leaf (1, 2))
 9                       (Node
10                        (Leaf ((3, 4), (5, 6)))
11                        (Leaf ((7, 8), (9, 0)))) :: Tree Integer
12    in
13      puti (size t1) (\ v -> v)
```

Figure 6.5: Polymorphic recursive example program

6.1.2 Polymorphic Recursion

One complication of our dynamic monomorphization approach arises in the presence of polymorphic recursion. Polymorphic recursion means that a recursive function calls itself (or other recursive functions in the same group of mutually recursive functions) with different types than those with which it was called. As an example, see the program in Figure 6.5, which defines both a polymorphic recursive data type `Tree a` (lines 1–2) and a polymorphic recursive function *size* (lines 3–6), which calculates the number of data elements in the tree. The program defines a tree (lines 8–11) and writes the tree's size (10 in the example) to the output (line 13).

The function `size` has the type annotation `Tree a -> Integer`, because our prototype implementation can only infer polymorphic recursive types in the presence of type annotations. This is similar to Haskell, which also supports polymorphic recursion for type-annotated functions.

The data type definition means that a value of type `Tree a` is either a leaf, which carries a value of type `a`, or an internal node with two children of type `Tree (a, a)`. That means that the leafs on each level have the same number of data elements, and that a leaf at level $n + 1$ has twice as many data elements as a leaf at level n. The tree constructed in the example program in Figure 6.5 is shown in Figure 6.6.

Polymorphic recursive data types similar to the one presented here can be used to type balanced and/or complete binary trees, for example.

When our monomorphization technique from the previous section is used for specializing polymorphic recursive functions, the following problem occurs: for each recursive call, which is a type application of a new type, a new specialized version of the function is created. The example program above would lead to three specializations for the `size` function: for the types `Integer`, `(Integer, Integer)` and `((Integer, Integer), (Integer, Integer))`. Of these types, the latter two have the same machine representation, which is a pointer to a heap allocated structure (a

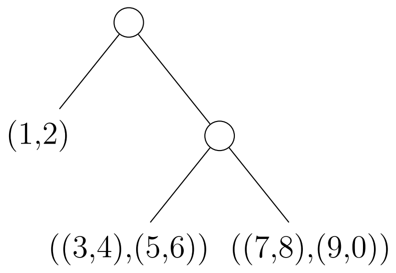

Figure 6.6: Polymorphic recursive example tree

tuple). Same machine representations lead to the same machine code, so code buffer space is wasted. The first type is represented as an unboxed integer value. Therefore, it should suffice to generate two specialized versions, one for unboxed integers and one for tuples.

We have solved this problem by grouping types with the same machine representation into equivalence classes. The specializer only creates new versions of polymorphic functions for new types when no specialization for the equivalence class of that type has been generated. The example program in Figure 6.5 then leads to the desired two specializations.

We currently distinguish three equivalence classes:

32-bit word Unboxed 32-bit value. Members of this class are not interpreted by the garbage collector. The built-in integer type and function pointers are in this class.

64-bit floating-point Floating-point values have special register passing conventions. Otherwise, they are also not interpreted by the garbage collector.

32-bit pointer Pointer to a heap-allocated record. These are traced by the garbage collector and must point to a header which defines the record layout.

When needed, this set of classes can easily extended, e.g. with 32-bit floating-point or 64-bit or 128-bit integer values.

Polymorphic recursive data types like the type `Tree a` above are handled in the same way by distinguishing members of the given equivalence classes (see Section 6.2).

Note that because of our support for polymorphic recursion, call sites of polymorphic functions can not be patched to jump to the monomorphic code like we have done for incremental compilation in the previous chapter. The problem is that one call site can possibly refer to calls with different concrete type parameters, so that the code which is generated for each call may expect different calling conventions and data layout. Therefore, no patching is done for monomorphized code and we rely on inline-caching for efficiency.

6.1.3 Type Classes

Type classes in Haskell (Hall et al., 1996) have originally been developed in order to tackle the problem of ad-hoc polymorphism (also called overloading) in a statically-typed language with type inference in a systematic way. The idea is to define classes of types by specifying a set of functions on these types, and then to add types to the classes by giving specific instances of these functions (also called methods). As an example, consider the class `Eq a`, which is the class of all types `a` that define equality. By giving instances, such as the instance `Eq Integer` for the integers, the type class gets populated. Note that in order to define an instance, a type-specific equality function (in the example for integer numbers) needs to be written.
In Haskell, the (simplified) definition of the `Eq` class looks similar to the following:[2]

```
class Eq a where
  eq :: a -> a -> Bool
  neq :: a -> a -> Bool
  neq x y = not (eq x y)
```

The type signatures for `eq` and `neq` declare them as methods, and the last line provides a default method for `neq`. Because of this default method, it is not necessary for all instances of `Eq` to provide a definition of this function, as it can be taken from the class. An instance for type `Bool` is given next:

```
instance Eq Bool where
  eq True True = True
  eq False False = True
  eq _ _ = False
```

Here, the method `eq` for type `Bool` is defined by pattern matching over the constructors `False` and `True`. When the function `neq` is used with boolean arguments, it will use the definition from the type class above, wich in turns calls the `eq` function for `Bool` and negates its result.
In our language, user-defined type classes are not supported. Some operators which can be found in the Haskell type classes `Num` ($+$, $-$, $\times$, $/$, negation) and `Ord` ($<$, $>$, $=$, $\neq$, $\geq$, $\leq$) are instead built into the compiler, as described in Section 4.3.2.
Most of the machinery for overload resolution is already provided by function specialization: when type values are substituted for type variables, the type annotations on overloaded integer literals and polymorphic primitive operators are instantiated with concrete types. The code generator is therefore given fully type-annotated monomorphic code. Integer literals which are annotated with the type `float`, for example can then be coerced to float literals during code generation, without any costly run-time type conversions.
The only drawback of this approach is that error messages are delayed: some type errors cannot be found at type inferencing time, but only later, when a polymorphic function is called at a particular type and its code is generated for the first time.

[2]Haskell uses infix operators instead of `eq` and `neq`.

Extension of our intermediate language and type system to support type classes is a topic of future work (Section 9.2.3). Combining type passing, cases on type constructors, optimization and dynamic specialization would lead to elimination of all overhead related to type classes.

6.2 Data Type Specialization

Similar to the specialization of polymorphic functions in the previous section, polymorphic algebraic data types can also be made monomorphic at run-time. The specialization algorithm from Section 6.1.1 only substitutes types for type variables, it does not state anything about the representation of constructor terms. When compiling for a real machine, these representation decisions have to be made. In traditional implementations of polymorphism, polymorphic values are represented in a standard form, normally as pointers to memory blocks which have the correct size to hold values of the used type. When functions are monomorphized, there is no need for a common representation, instead the representation can be tailored to the data type under consideration. The advantages are similar to the case for functions: individual data values can make better usage of machine-specific representations and access time to data elements can be reduced.

One disadvantage for using this technique in lazy functional languages is that it must be combined with strictness analysis or programmer-supplied strictness annotations in order to discover opportunities for optimization. Without strictness information, the optimizer has to assume that each element of an algebraic type has the additional value $\perp$, so that unboxing cannot be performed without changing the behavior of the program.

Kafka, as a strict language, does not suffer from this problem, but further investigation of dynamic specialization and laziness is also a topic for future work.

Modern processors are very sensitive to the layout of compound data in memory and to access patterns to these data elements. Large, but slow memories and multiple layers of cache memory of limited size require optimization of storage layout for the cache architecture of the machine on which a program is running. With a dynamically compiling system, data structures can be optimized for the executing machine when a program is run.

Of course, not all such optimizations can be expressed in our high-level description, but only in a low-level language for expressing data layouts. Our implementation handles the ordering of data fields on the level of the CPS language, but alignment is done by the code generator on the fly.

Since performance of programs on modern processors depends greatly on the ability to make use of data and instruction caches, we focus on this aspect of data representation. While most optimizations on programs target instruction cache locality, data representations affect the locality of the data caches.

When specializing data structures, we need to take several aspects into account:

– Data structure elements should use their native encoding, to avoid any boxing and unboxing or tagging and untagging overhead.

- Values should be aligned for optimal access times. On some platforms, it may even be necessary to align them, e.g. on SPARC (Weaver and Germond, 1994), which raises a hardware exception for unaligned accesses.
- Data structure elements should be grouped together so that cache locality can be exploited. Often used elements should be put together, less frequently used elements can be put out-of-line. This is an optimization for future work (see Section 9.2.4).

6.2.1 Type-directed Representation Selection

First of all, we have to decide how to represent the different types of the source language.

- Values of the predefined integer type are represented as 32-bit signed words.
- Values of the predefined floating-point type are represented as 64-bit IEEE floating-point numbers.
- The predefined Boolean type is represented as if it were declared as the algebraic data type

  ```
  data Bool = False | True
  ```

  in the source program.
- The predefined unit type is represented as the integer 0. Theoretically, values of this type would never need to be explicitly represented, but doing so simplified our implementation.
- Closures are represented as pointers to heap records, where the first field is a header, the second a function pointer and the free variables are represented as if they were members of a user-defined constructor (see below).
- User-defined algebraic types are in principle represented as pointers to heap-allocated records (for exceptions, see below). The first field is a header, the second a small integer representing the variant (also called the *tag*). All other fields are represented as described in this section (recursively) and layed out as described in the next section.
- Tuples are also represented like user-defined product types, but without the need for a variant tag.

We have additionally implemented the following type-directed data representation optimization for algebraic types:

- Data types where no variants contains fields (enumerations) are represented as consecutive small integer values. This allows efficient dispatch on the constructors.

– Data types which have both constructors with and without fields use both unboxed and boxed representations: constructors without fields use unboxed small integer values, constructors with fields use pointers to allocated heap records. When a case distinction is made on values of this type, the unboxed variants are tested first, when the value does not correspond to an unboxed variant, it is safe to load the first word from the heap object the value points to. Dispatch then proceeds on the constructor tag stored in the heap-allocated record.

Heap pointers can always be distinguished from unboxed constructors because the latter are small integers which are different from valid heap addresses on most machines. On machines without unused ranges in the address space, data types with at least one variant with a field, all values of these types would need to be boxed.
The propagation of type information to the code generator allows to represent integer values as full machine words, without any need for tagging (as was used in the SML/NJ compiler, for example (Appel, 1992)). Using the type information, the code generator can construct precise tables describing which registers and stack slots contain possible pointers to heap cells and which do not.
The second step in choosing a representation is the layout of individual fields, which is described below. Fields of different sizes should be grouped such that there are no unnecessary gaps due to alignment. This requirement can be fulfilled using only size information of individual fields.

Examples

The boolean type

```
data Bool = False | True
```

is represented by the integers `0` and `1`.
The list type

```
data ListInt = Nil | Cons Integer ListInt
```

(which results from specializing the type `List a` for type `Integer`) is represented by the integer `0` for the `Nil` variant and a tagged pointer to a heap-allocated record of 3 words: a tag, the integer field and a pointer to the rest of the list. Note that the tag for this data type is not really necessary, because there is only one boxed variant, but our implementation does not currently do this optimization. The type `Maybe a = Nothing | Just a` can be represented similarly.
Figure 6.7 shows possible data representations for values of type `List Integer` and `List Float`. In both cases, the constructor `Nil` can be represented by the integer 0. The `Cons` constructors should have different layouts for the `Integer` and `Float` instantiations. The `List Integer` object (arbitrarily) has the data value in the second slot (after the constructor tag), whereas for the `List Float` object, the data

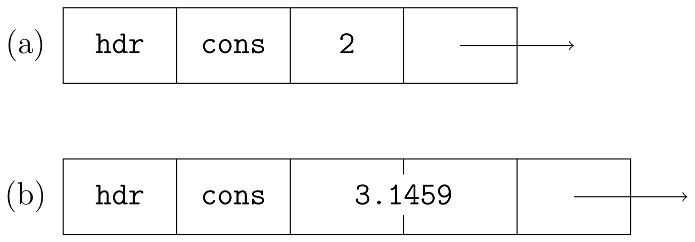

Figure 6.7: Possible data representations for (a) `List Integer` and (b) `List Float`

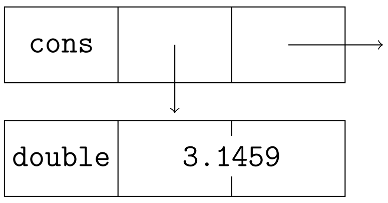

Figure 6.8: Common data representation for `List Float`

field occupies two words. Since the header and the tag are eight bytes long, the floating-point field follows directly, as it is 8-byte-aligned.
Consider the representation which is used in most existing functional language implementations, shown in Figure 6.8. (The header field is omitted.) Here, the double value is boxed to achieve a uniform layout of heap objects. The overheads are increased memory requirements (thus increased allocation and garbage collection time) and additional indirections to access the data.

6.2.2 Data Layout Algorithm

We have designed and implemented a simple data layout algorithm for structures allocated on the heap. The goal of the algorithm is to determine good layouts for algebraic data types, tuples and closures. The results should enable the following:

– Fields should be aligned at the boundaries required for their machine types.
– Cache locality should be improved by reducing memory requirements.

The algorithm proceeds in three steps:

1. For each field, its original index in the record, its size and alignment is calculated.
2. The fields are sorted by decreasing size (using a stable sort).
3. When a field is not aligned now at its natural alignment boundary, dummy fields of word size are inserted before the field until it is aligned properly. Rhis step is performed for all fields from first to last.

This information is used for the following tasks. The sizes of the fields and their alignment must be known, because when creating a constructed value, a tuple or a closure, the size to be allocated is the sum of the aligned sizes of all fields. The original field indices are then required to map the fields in the intermediate representation to the memory locations in the optimized layout. Similarly, on projections from tuples and closures, the projection index must be mapped to an offset in the heap record. Case expressions, which are used to extract fields from algebraic data types, also have to fetch the fields in order to move them to the stack or to registers.

The resulting code is very efficient, because all references to fields are mapped to constant offsets in the machine codes. Due to the specialization of polymorphic data types, the implementation is free to choose any memory layout it wants. In our prototype implementation, the optimized memory layout can be switched off using a command line option, in order to measure its performance impact and to experiment with alternative layout algorithms.

In Section 9.2.4, we propose some improvements to this basic layout strategy, for example grouping of fields which are often used together.

Chapter 7

Implementation

The concepts and techniques presented in the previous chapters have been implemented as a prototype system called Kafka. In this chapter, we give details on several aspects of the implementation and discuss the optimizations used for improving the intermediate code. Most of the optimizations are adaptations of classical transformations for functional languages adapted to our intermediate language and specialization techniques. We also give a concrete description of the implementation, about its size, its implementation language and used libraries.

7.1 Implementation Outline

The prototype system was implemented in Haskell (Peyton Jones, 2003) and developed using the Glasgow Haskell Compiler (GHC) (GHC Developers, 2008). The project consists of about 10 300 lines of Haskell code, including the parser, type checker, optimizer, code generator, virtual machine and stand-alone compiler.
The system runs on IA-32 (or x86 architecture) computers as described in Intel's developer manual (Intel Corporation, 2006). It has so far only been tested on GNU/Linux systems.
Some of the timing results in Chapter 8 have been gathered using the `rdtsc` instruction of the Pentium processor. This instruction was accessed using the small Haskell library rdtsc (Grabmüller, 2008), which is a small wrapper around a C function which invokes the `rdtsc` instruction using inline assembler.
Except for the rdtsc library, the complete prototype system is written in pure Haskell. We emphasize this fact because it shows that Haskell is well-suited for both writing system-level programs (such as the code generator or garbage collector) and expressing high-level algorithms, such as Milner-Damas-style type inference and complicated program transformations.

7.1.1 Front End

The parser was implemented using the Parsec parser combinator library (Leijen and Meijer, 2001).

The type checker implements Milner-Damas-style type inference and uses an extension of algorithm M (Lee and Yi, 1998), which includes a dependency resolution algorithm for identifying mutually recursive functions. The output of the type checker is the fully typed language which is described in Chapter 4 as the “source language”.

7.1.2 Conversion and Optimization

The result of typechecking is the input to the CPS conversion described in Chapter 4. CPS terms are optimized using several optimizations described below, then closure-converted (see also Chapter 4), optimized again and fed to the code generator. The reason for two rounds of optimization is that the closure conversion introduces opportunities for optimization which are not present before this phase, for example projections from known closures.

7.1.3 Code Generation

Machine code generation is performed using the Haskell library Harpy (Grabmüller and Kleeblatt, 2007). This library allows the generation of machine code directly into a memory buffer and provides label management, patching of forward references, checking for code buffer overflow and a disassembler. It also defines a convenient domain-specific language for assembler programming in Haskell, using type classes for supporting various addressing modes. The Harpy library has about 7 600 lines of Haskell source code (including the disassembler, which alone amounts to 2 700 lines).

7.1.4 Run-time System

The run-time code generator is part of the run-time system. Whenever a delay expression is executed or a polymorphic function is called, the code generator is invoked. Its main purpose is to generate monomorphic code and to place new delay expressions, but in theory it could do any useful transformation on the program it is to process. We have not exploited this possibility yet.

The garbage collector is a simple two-space copying collector (Cheney, 1970). At the beginning of each function, a check for heap overflow is generated. When the check fails, the values of all registers are saved to the stack and two bitmaps are passed to the collector: one specifies which registers hold live pointers into the heap, and the other specifies the live pointers on the stack. The collector than copies the objects pointed to by these pointers (and some globally known pointers) into an empty memory area. Then all live pointers recursively contained in these objects are copied, until the complete live object graph is in the new memory area. The old area can then be reused in the next collection cycle. In order to correctly interpret the fields of heap objects, the first word of each object is a pointer to a descriptor, which also contains a bitmap identifying pointer fields.

7.2 Optimizations

Our prototype system implements a number of standard optimizations, because otherwise the results of measuring specialization effects cannot be compared to other language implementations, which all contain optimizations similar to the ones we describe here.
We will only briefly describe the optimizations, but when specialization or dynamic compilation has an effect on some optimization, we mention it below in their descriptions.
In this section, we will use example programs in the source language instead of the CPS language, in order to maintain readability.
The following optimizations are performed during compilation:

Function inlining User functions used only once are inlined at their call sites. Inlining of user functions is also important in implementing other optimizations, for example uncurrying (see below). Therefore, functions introduced by other optimizations can be marked for inlining, so that they are forced to be inlined even when called more than once. Inlining is a standard (but important) optimization for all optimizing compilers, but especially for functional languages, where most functions are small and function calls are frequent.
Inlining is affected by our specialization technique. Polymorphic functions are translated to functions which receive types as parameters. When such a function is inlined, the actual parameters (which are type variables) must be substituted in terms as well as in types. This is necessary because all intermediate terms are annotated with types, and types can appear at the term level for applications of type functions.

Continuation inlining Continuations used only once are inlined at their call sites. This is especially important for the continuations which are generated for the alternatives of a case expression, because these are always called once. Continuation inlining must includespecial handling of type parameters for the same reason as function inlining.

Case idiom recognition Case expressions often scrutinize boolean variables which are the result of some primitive comparison operator. These are converted to a special form of primitive conditional expressions. The reason is that most instruction sets allow conditional jumps only on the result of some comparison instruction – this does not work well with comparisons on algebraic data types representing boolean values, because the boolean results must be represented in order to perform a case analysis on them. This optimization has also been described by Appel (1992, p. 75).

Uncurrying Curried functions (that is, nested λ abstractions in the source language) are inefficient when applied to several arguments at once, because for each abstraction, an intermediate closure is created which is immediately called. Therefore, the uncurrying optimization extends nested function definitions into functions

which take several arguments at once, and adjusts the call sites to provide all actual arguments together. We use the uncurrying method described by Appel (1992, p. 76), which converts curried functions to uncurried ones and adds curried wrapper functions. These wrappers are then inlined at their call sites, effectively uncurrying the function call.
As mentioned by Tarditi (1996), Appel's transformation is underspecified and does not necessarily result in uncurrying of functions with more than two arguments. Like Tarditi, we have fixed this problem by applying the uncurrying optimization in the order from outer to inner expressions.
Uncurrying is affected by the specialization on polymorphic functions, because we cannot merge a type abstraction with a value abstraction when the type of the latter depends on the former parameter. Consider the following example:

```
let id = \ a: *. \ x: a. x
in ...
```

If we uncurry this definition, we get the following function:

```
let id = \ (a: *, x: a). x
in ...
```

The problem is that when generating machine code for such a definition, we do not know the representation of the type `a`, so cannot correctly calculate whether the value of `x` will be placed in an integer register, a floating-point register or on the stack. Therefore, our implementation avoids uncurrying in this manner and all functions receive either type or value parameters. Independent type and value parameters could be uncurried together, but in our test cases these functions were extremely rare, so this is probably not worth the effort.

Common subexpression elimination (CSE) The common subexpression elimination phase replaces values which are calculated several times by the variables to which the values have already bound. Our CSE is flow-insensitive, as it does not take control flow around loops into account. Thus, it only eliminates common subexpressions when one lexically dominates the other.

Dead variable elimination Unused bindings of values, projections, continuations, functions and closures are removed from the program. The results of primitive operations are only removed when the operation cannot have any side effect (such as raising an exception).

Constant folding Primitive operations on constants are calculated at compile time, except when the operation would cause an exception at run time (for example, division by zero). Constant folding often enables dead variable elimination for the operands.

Projection elimination Projections from known tuples or closures are performed at compile time. In some cases, when the projection is the only use of a tuple or closure value, the construction of the respective object can be removed by dead variable elimination (see above).

Case elimination Cases on known values are reduced at compile time. This optimization is important because inlining often creates case expressions which scrutinize known values, so this optimization not only saves run-time (by eliminating a test and a branch instruction), but also removes unreachable code. This can improve compile times significantly, which is always a concern in dynamically compiling systems.

Floating-point register assignment Since the IA-32 architecture provides so few integer registers, where most of them are used to hold the virtual machine state in our implementation, and others have fixed uses for parameter passing, there is not much point in sophisticated register allocation for these registers. Floating-point registers, on the other hand, are plenty, so we have implemented a simple register assignment scheme which tries to hold as many floating-point intermediate results as possible in registers.

Inline caching Specialization of polymorphic functions requires that each call to a polymorphic function checks whether a specialization for the actual type parameters already exists. Since this check is rather expensive (in our implementation, it is written in Haskell, and the transition from JIT-compiled code to Haskell code is costly[1]), we have implemented an optimization called inline caching. In object-oriented systems, this optimization is used to avoid method lookups (Detlefs and Agesen, 1999). The idea is to generate a short instruction sequence which tests whether the actual type parameters are equal to the ones given the last time the function has been called. If they match, the code from the last specialization can be used, otherwise, the specializer is invoked. After specialization, the values tested in the code sequence are overwritten by the specializer by modifying the testing code: constants in the instruction stream which refer to types are overwritten with the new types, and the jump instruction at the end of the inline code is overwritten to jump to the newly generated code. Since the short in-line sequence of code is very fast, the overall execution time is reduced significantly.

Recursion optimization Without any precautions, self-recursive calls of polymorphic functions always go through the specializer (or at least through the inline caching mechanism). In addition to the overhead caused by checking whether a specialized

[1]Transitions from Haskell code to compiled code and vice versa go through Haskell's Foreign Function Interface (FFI), which involves checking for interrupts and whether garbage collection is necessary, marshalling and unmarshalling of arguments and results, etc. This has non-negligible overheads.

version exists, the type parameters have to be passed and this results in curried functions (see above in the description of uncurrying on page 91). Recursion optimization modifies polymorphic recursive functions so that the function body is changed into a let expression which binds a new local function to the old function body. The body of the let calls the local function. Since the local function is monomorphic, no specialization overhead occurs. As an example, consider the following recursive function, which calculates the length of a list:

```
let len = \ a:* -> \ l: List a -> case l of
                                    Nil -> 0
                                    Cons x xs -> len a xs + 1
```

It is converted to the following:

```
let len = \ a:* -> \ l: List a ->
             let len’ = \ l: List a -> case l of
                                          Nil -> 0
                                          Cons x xs -> len’ xs + 1
             in len’ l
```

The local function can be compiled to a loop without any specialization overhead. This optimization has been called “Loop headers in λ-calculus or CPS” by Appel (1994), because it is related to loop header introduction in the compilation of loops for imperative languages.
For polymorphic recursive functions, this optimization is not valid, because it assumes that data representations for recursive calls at one call site are always the same. Therefore, we do not apply this optimization for polymorphic recursive functions.

Data layout optimization The fields of tuples, closures and constructed values of algebraic data types are rearranged in order to align them properly. For example, 64-bit floating-point fields are always aligned at an 8-byte boundary. Our data layout algorithm is defined in Section 6.2.

Data alignment optimization When the virtual machine is run, the alignment of allocated heap data (tuples, closures and constructor applications) can be specified on the command line. The decision of an optimal alignment could also been based on the executing processor and its memory organization, but this has not been investigated for our prototype. In contrast to the previous optimization, this one refers to alignment of heap objects in memory, not to the alignment of individual fields in one object.

Code alignment optimization When the virtual machine is run, the alignment of jump targets in the generated machine code can be specified on the command line. Intel Corporation (2005) recommends to align jump targets at 16-byte boundaries, but for the programs we tested, this option did not affect run-time at all, probably because our benchmark programs can be kept completely in the first-level cache.

Chapter 8

Experimental Results

In order to establish the practicability of our approach, we have conducted a number of experiments for measuring and comparing the performance of our prototype implementation. The experiments investigate the impact of compiler optimizations, the performance of the produced code relative to other implementations of functional programming languages and the compile times of our dynamically compiling system. This chapter describes the test methods, the hardware used, the benchmark programs and presents the measured results.

The results show that our prototype has competitive performance on several small benchmarks, when compared to mature statically compiling implementations. For one program of realistic size, performance is within a factor of ten when compared to an optimizing Haskell implementation, but only requiring half of the time to compile the program.

8.1 Test Methodology

We have first measured the performance of our prototype implementation in order to test the effect of the implemented optimizations. All benchmarks were performed multiple times: first with all optimizations switched on, then with all optimizations switched off. After that, we have tested the effect of twelve optimizations by individually switching them off.

In a second set of measurements, we have written the benchmark programs in various other languages (functional, imperative and object-oriented) and compared the performance of our prototype to other language implementations.

The third experiment consisted of porting a larger benchmark program from the literature to our prototype and comparing it to several other languages.

All benchmarks were done on a Intel Pentium 4 CPU, running at 2.4 GHz with 512 MB of RAM and a second-level cache of 512 KB. The operating system used was Debian GNU/Linux 4.0 with a Linux kernel version 2.6.18.

Each benchmark program was run three times, and the best result was used. Timings were obtained using the `time` built-in command of the `bash` shell, and user times are reported.

Name	**Description**
nfib	nfib function (integer version)
rfib	nfib function (floating-point version)
pfib	nfib function (both integer and floating-point version)
nlenrev	construct a list of integers, reverse it, calculate length
rlenrev	construct a list of floating-point values, reverse it, calculate length

Table 8.1: Common benchmark programs

The benchmarks in the different languages have been written in a way to maximize their performance. For example, when special operators are available for small integers, these have been used, and we use built-in list functions when available. In several cases, we have also measured other program variants, so that the comparison between different languages is easier.

8.2 Benchmark Programs

In order to compare the performance of our prototype implementation to other language implementations, we have written a suite of small benchmark programs. The benchmarks can be put into three different groups:

– Function-call intensive programs with a high amount of basic arithmetic operations.

– Data-structure intensive programs which create large dynamic data structures and traverse them.

– A realistic benchmark program.

In the first category, called *fib*, we have implemented several versions of the `nfib` program from the nofib benchmark suite for Haskell (Partain, 1993). The nfib function counts the number of function calls required to calculate the Fibonacci number of the input n. In the second group we have implemented the *lists* benchmarks, which work on linked lists. And finally, the *trees* benchmark constructs and searches binary search trees. The last group contains only one program: the floating-point intensive, realistic, Pseudoknot benchmark.
Table 8.1 summarizes the common benchmarks, which have been implemented in all languages, whereas the special benchmarks in Table 8.2 have only been written for languages which support them, or in order to get additional insight into the actual benchmark results.

fib The programs in this group are aimed at testing function call efficiency and the implementation of basic arithmetic operations. The first version of the *fib* benchmark is called *nfib* and works with native machine integers. We have made sure (by

Name	**Description**
ffib	32-bit floating-point version of rfib
nfibmod	modular version of nfib
pfibmod	modular version of pfib
nlenrevnogc	nlenrev without garbage collection
rlenrevnogc	rlenrev without garbage collection
tree	search tree construction and searching
pseudoknot	float-intensive realistic benchmark

Table 8.2: Special benchmark programs

using type annotations or special arithmetic instructions when necessary), that all implementations work on integers not larger than a machine word, so that we can expect the best possible performance for each implementation. The *rfib* version works with floating-point numbers, which are implemented as 64-bit IEEE floating-point numbers in all implementations. The *pfib* benchmark calculates both with integer and with floating-point numbers, and uses the same, polymorphic implementation for both uses. This version tests the efficiency of polymorphic functions and overloaded arithmetic.
Additionally, in order to evaluate several aspects of some implementations, we have written three special versions of the *fib* theme: *pfibmod* is the same as *pfib*, but with the polymorphic function in a separate compilation unit. *Nfibmod* is the same for the integer version. The *ffib* program works on 32-bit floating-point numbers instead of 64-bit values.

lists The *nlenrev* and *rlenrev* benchmarks are data structure intensive programs which each construct a list of one million elements, reverse the list and then take the length of the result. Similar to the *fib* benchmarks, *nlenrev* works on integer lists and *rlenrev* on lists containing floating-point values.
Since garbage collection is a weak spot in our implementation, we have made additional experiments where the heap size allocated on startup is large enough to that no garbage collection has to take place during the benchmark run. These benchmarks are called *nlenrevnogc* and *rlenrevnogc*.

trees The tree benchmark creates two unbalanced binary search trees, one small tree with mixed contents and one created by repeatedly inserting elements of a sorted 10 000-element list. The trees are searched for values which are in the trees and for one value which is not. The *tree* test program also belongs to the data structure intensive benchmarks.

pseudoknot This is a floating-point intensive realistic program which solves a problem from molecular biology. It is described in detail in Section 8.5 below.

Code	**Optimization**	**Description**
ci	Case idioms	Recognize cases on comparison results
if	Inline funs	Inline user functions
ic	Inline conts	Inline continuations
il	Inline	Inline both functions and continuations
uc	Uncurrying	Uncurry functions and continuations
cs	CSE	Common subexpression elimination
dv	Dead Vars	Remove unused bindings
cf	Const Folding	Folding of operators, projections etc.
ly	Layout	Data structure layout
fp	FP Regs	Optimize floating-point register assignment
rc	Recursion	Create local versions for recursive functions
ca	Inline Cache	Inline caching for polymorphic code

Table 8.3: Measured optimizations

8.3 Effect of Implemented Optimizations

The effect of optimizations is quite difficult to predict, mainly because different optimizations interact in complicated ways. Additionally, optimizing transformations are in general not commutative, so the order in which they are carried out can affect the performance of the resulting program significantly.
So in order to measure the effects of our implemented optimizations, we have compiled and run the benchmark programs with various optimizations switched off, and compared the running times with the times for a run with all optimizations switched on and without any optimizations, respectively. In Table 8.3, the optimizations measured and their abbreviations (to be used in the result tables) are summarized. All optimizations are described in detail in Section 7.2.

Results

Tables 8.4 and 8.5 summarize the results. The "O0" lines reports the run times when all optimizations are switched off (command line option `-O0`), whereas the "O2" lines denote the results for fully optimized programs (command line option `-O2`). Both lines are highlighted in the tables for better readability. All other lines are showing the results for the benchmarks when one particular optimization is turned off (the abbreviations for the optimizations are given in Table 8.3). The second column gives the absolute run time in seconds and the third column gives the run time relative to the "O2" line. The names of the benchmarks are the names from Tables 8.1 and 8.2, but with the suffix "opt".
Longer run times for a disabled optimization mean that the optimization is good, because switching it on reduces run times.
The run times include parsing, type checking and various transformations and opti-

mizations, such as CPS conversion, closure conversion, all optimizations not switched off, and machine code generation. Since the benchmark programs measured in this section are small, most of the run time is used up by running the code, not by compiling it.
For the benchmark pfibopt, we cannot report results for optimization level `-O0`, because the benchmark allocates too much memory to successfully terminate. For some other benchmarks, the heap size had to be increased in order to run them successfully. In the case of *nlenrev* and *rlenrev* (O0), 400 MB of heap were allocated, for the other *list* benchmarks 70 MB. All *tree* benchmarks were performed with 200 MB of heap. The *nfib* benchmarks ran with the default heap size of 1 MB.

General Remarks

The completely unoptimized programs are 2.7–12 times slower than the fully optimized programs.

nfib For the floating-point version, the difference between the unoptimized and the fully optimized program is less than for the integer version. This is probably due to more memory traffic in the floating-point code. Switching off individual optimizations can improve run times by up to 3%.

lists The difference between integer and floating-point code is similar to the nfib benchmarks.

trees For this benchmark, we have set the input size to 4000, because the unoptimized programs use more memory than is available in the test machine.
The *tree* benchmark seems to contain functions for which uncurrying has a negative effect. Therefore, switching off uncurrying or function inlining (which is required by uncurrying) makes the program run much faster, up to 40%.

Remarks on Individual Optimizations

The case idiom optimization (ci) has an effect of about 1–2%, and sometimes it slows down the program.
Switching off inlining functions (if) changes run times by a few percent (both up and down), and has more significant effects for the *lists* (slower by about 100%) and *tree* benchmark (faster by about 40%). This makes it very difficult to recommend it.
Inlining continuations (ic) is very important for our implementation, because many continuations are introduced which can be removed when e.g. functions are inlined. Therefore, switching this optimization off gravely affects performance.
Switching off inlining (il) in general (both user functions and continuations) has similar effects than inlining of continuations, which stresses that continuation inlining is more important than function inlining.

nfibopt 40		
impl	time	rel
cs	8.405	0.98
cf	8.437	0.99
if	8.465	0.99
ca	8.477	0.99
fp	8.489	0.99
uc	8.505	1.00
ly	8.529	1.00
O2	8.545	1.00
rc	8.565	1.00
ci	8.705	1.02
dv	8.705	1.02
ic	25.418	2.97
il	25.434	2.98
O0	25.806	3.02

rfibopt 40		
impl	time	rel
cf	11.725	0.97
cs	11.773	0.98
if	11.865	0.98
ca	11.901	0.99
ly	11.937	0.99
ci	11.953	0.99
rc	12.025	1.00
dv	12.033	1.00
O2	12.073	1.00
fp	12.085	1.00
uc	12.089	1.00
ic	32.162	2.66
il	32.878	2.72
O0	32.898	2.72

pfibopt 40 40		
impl	time	rel
O2	20.229	1.00
cs	20.253	1.00
ly	20.325	1.00
cf	20.425	1.01
if	20.541	1.02
fp	20.561	1.02
ca	20.593	1.02
ci	20.665	1.02
uc	20.801	1.03
dv	20.817	1.03
il	58.504	2.89
rc	58.504	2.89
ic	58.536	2.89

Table 8.4: Benchmark results: effects of various optimizations (part 1), benchmark pfibopt: measurements for “O0” missing (see text)

nlenrevopt 1000000		
impl	time	rel
cf	1.372	0.99
fp	1.380	0.99
O2	1.392	1.00
ly	1.396	1.00
cs	1.404	1.01
ca	1.440	1.03
ci	1.444	1.04
dv	2.084	1.50
uc	2.660	1.91
if	2.708	1.95
il	7.052	5.07
ic	7.156	5.14
rc	8.129	5.84
O0	10.025	7.20

rlenrevopt 1000000		
impl	time	rel
O2	3.576	1.00
ci	3.672	1.03
cf	3.676	1.03
cs	3.692	1.03
ca	3.720	1.04
ly	3.720	1.04
fp	3.744	1.05
dv	4.424	1.24
uc	7.324	2.05
if	7.440	2.08
O0	9.521	2.66
ic	14.025	3.92
il	14.161	3.96
rc	15.653	4.38

treeopt 4000		
impl	time	rel
if	1.164	0.60
uc	1.304	0.68
O2	1.924	1.00
dv	1.928	1.00
ca	1.932	1.00
ly	1.940	1.01
ci	1.948	1.01
cf	1.956	1.02
cs	1.968	1.02
fp	1.984	1.03
ic	2.656	1.38
il	2.668	1.39
rc	2.788	1.45
O0	23.233	12.08

Table 8.5: Benchmark results: of effects various optimizations (part 2)

The uncurrying optimization (uc) is sometimes an advantage, sometimes a disadvantage. For the *fib* benchmarks, which only have functions of one value argument, there is no opportunity for uncurrying, so the compile-time effort is wasted. As mentioned above, uncurrying gives very bad results for the *tree* benchmark.
Common subexpression elimination (cs) affects run time by up to 3%, but for most benchmark there is no effect at all or even a slow down due to compile time costs.
Dead variable elimination (dv) does not give problems for these benchmarks and can affect run time by up to 50% for the *nlenrevopt* benchmark.
Constant folding (cf) makes a difference of a few percent, in both directions.
Layout optimization (ly) has mostly an effect of 4% for *rlenrevopt*, since this benchmarks has data structures (lists of floating-point values) which benefit most from this optimization which aligns these values on 8-byte boundaries.
Optimization of floating-point registers (fp) has a small impact of 1–2% for most benchmarks. For some benchmarks, run times improve when this optimization is switched off. This is probably due to reduced compilation times, since several of the benchmarks do not use floating-point values at all.
Recursion optimization (rc) is the most effective single optimization. The reason is that, for polymorphic functions, it avoids a lookup in the specialization cache on each recursive invocation.
The effect of switching off inline caching (ca) does not have much effects for the benchmarks except for *nlenrevopt* and *treeopt*. The reason is that for the *nfib* benchmark, the recursive calls are monomorphic because of recursion optimization, so that the inline cache is not used much.

8.4 Comparison to other Implementations

The common benchmark programs have been written in Kafka, Haskell (Peyton Jones, 2003), Standard ML (Milner et al., 1997), C/C++ (Kernighan and Ritchie, 1988; Stroustrup, 1986), Scheme (Kelsey et al., 1998), Java (Gosling et al., 2000) and Opal (Pepper, 2003).
The special benchmarks have not been implemented in all languages, see below for details.
We have used the Glasgow Haskell Compiler (GHC Developers, 2008) for the Haskell programs, MLton (MLton Developers, 2006) for the Standard ML programs, Bigloo (Serrano and Weis, 1995; Bigloo Developers, 2008) for the Scheme programs, the Opal Compilation System (Opal Group, 2004) for the Opal programs, the GNU compiler collection C and C++ compilers (GCC Developers, 2008) and the Sun Java development Kit (Sun, Inc., 2008) compiler and virtual machine for Java. The Kafka programs are of course run by our prototype implementation.
Table 8.6 lists the versions of the language implementations.
All compilers have been used with optimizations enabled, the actual command line arguments are listed in Table 8.7.
One important fact to keep in mind when interpreting the results in the next section

Language	**Implementation**	**Version**
Kafka	Kafka prototype system	0.1
Haskell	Glasgow Haskell Compiler (GHC)	6.8.2
Scheme	Bigloo	2.8c
Standard ML	MLton	20061107
Opal	Opal Compilation System (ocs)	2.3k
C/C++	GNU Compiler Collection (gcc)	4.1.2
Java	Sun Java Development Kit	1.5.0

Table 8.6: Implementation versions

Implementation	**Optimization options**
Kafka	—
GHC	`-O2`
Bigloo	`-O6`
MLton	—
Opal	`opt=full`
C/C++	`-O4`
Java	—

Table 8.7: Command line options for optimization

is that the various language implementations are not completely equivalent. For example, some languages (such as Standard ML and Opal) perform overflow checking on arithmetic, whereas the others do not. This is a disadvantage for the former implementations. Also note that the Haskell implementations are lazy, and all other implementations are call-by-value. Nevertheless, we think that even if it is a kind of apple-to-oranges comparison, the numbers presented next are an indication of the implementations' performance in "real life".

Results

Tables 8.8 to 8.10 give the results of our experiments. In each table, the first line gives the name of the benchmark together with the input size. For example, the *pfib* benchmark in Table 8.8 has been run with an input of 40 for the integer function, and an input of 40 for the floating-point function as well.

Of course, since all benchmarks described here are small benchmarks which at best correspond to small kernels of real applications, the results do not generalize to real programs, they merely give an intuition of the performance under varying circumstances.

We will now discuss the results in detail.

nfib 40		
impl	time	rel
bigloo	1.024	1.00
gcc	1.036	1.01
ghc	2.772	2.71
java	3.120	3.05
mlton	3.192	3.12
kafka	8.429	8.23
opal	10.349	10.11

rfib 40		
impl	time	rel
gcc	2.756	1.00
java	4.676	1.70
mlton	8.329	3.02
kafka	11.805	4.28
ghc	18.653	6.77
bigloo	30.378	11.02
opal	55.071	19.98

pfib 40 40		
impl	time	rel
gcc	3.848	1.00
mlton	11.057	2.87
kafka	20.337	5.29
ghc	22.325	5.80
bigloo	31.666	8.23
opal	122.944	31.95

pfibmod 40 40		
impl	time	rel
ghc	140.817	1.00

ffib 40		
impl	time	rel
opal	18.441	1.00

nfibmod 40		
impl	time	rel
gcc	1.024	1.00

Table 8.8: Benchmark results: nfib

nlenrev 1000000		
impl	time	rel
mlton	0.040	1.00
opal	0.104	2.60
ghc	0.200	5.00
bigloo	0.208	5.20
java	1.184	29.60
kafka	1.452	36.30

nlenrev 1000000 nogc		
impl	time	rel
bigloo	0.052	1.00
ghc	0.068	1.31
kafka	0.096	1.85
gcc	0.156	3.00
java	0.460	8.85

rlenrev 1000000		
impl	time	rel
mlton	0.056	1.00
ghc	0.188	3.36
bigloo	0.208	3.71
opal	0.208	3.71
java	2.548	45.50
kafka	3.644	65.07

rlenrev 1000000 nogc		
impl	time	rel
bigloo	0.060	1.00
kafka	0.092	1.53
ghc	0.100	1.67
gcc	0.144	2.40
java	1.040	17.33

Table 8.9: Benchmark results: lists

tree 10000		
impl	time	rel
mlton	1.764	1.00
kafka	3.204	1.82
ghc	4.764	2.70
bigloo	9.369	5.31

Table 8.10: Benchmark results: trees

Results for nfib

nfib For the integer version of the nfib function, it is remarkable that the Scheme version compiled by Bigloo is fastest. The main reason is that for the Bigloo version, we used explicit fixnum arithmetic. Nevertheless, the fact that Bigloo is faster than the C version is quite impressive, as Bigloo compiles to C and then invokes the C compiler. The GHC version also comes out quite fast, because the strictness analyzer detects that all computations can be performed strictly. GHC also performs specialization for the nfib function. Kafka is slower than most of the other implementations, but note that parsing, type checking, optimization and code generation are included in the timings for Kafka. Opal does not optimize function calls and simple arithmetic well, apparently.

rfib For *rfib*, the results are quire different than for the last benchmark. GHC and Bigloo apparently generate worse code for floating-point intensive programs. Even though GHC again performs specialization, its floating-point code performs worse then the integer code. MLton is a specializing whole-program compiler anyway, and seems to generate good floating-point code, too. Bigloo takes a performance hit because the implementation cannot use the faster fixnum arithmetic here. Kafka's handling of floating-point code by specialization pays off here, so we get good results. The Java results show how good just-in-time compilation can perform.

pfib For the polymorphic version, Kafka performs especially well because of the run-time specialization. The floating-point version of the polymorphic nfib function runs just as fast as for the *rfib* benchmarks. For most implementations, the times of the *pfib* benchmarks are approximately the sum of the *nfib* and *rfib* runs; only Opal is much slower, because it lacks function polymorphism: all overloaded functions and constants must be given as module parameters. For Java, we have no *pfib* implementation, because it lacks both polymorphic functions over primitive types and an efficient means for simulating them.

pfibmod We have made another version of the *pfib* benchmark for Haskell, in order to see how strictness analysis and type classes affect performance. In the *pfibmod* program, the nfib function is placed in a separately compiled module. The effect is dramatic: the modular version is about six times slower than the monolithic, because no automatic specialization is done. Without specialization and strictness analysis, laziness and the overhead of type classes cannot be avoided.

ffib Because of the bad performance of Opal for *rfib*, we have additionally measured its performance when using 32-bit floating-point values instead of 64-bit values. The run-time for *ffib* is roughly one-third of that for *rfib*, because 32-bit floating-point values are represented more efficiently in the Opal run-time system.

nfibmod As a small additional check, we have made a modular version of the *nfib* program for the C compiler, in order to see what effect modularity has for this implementation. Interestingly, the modular variant is *faster* than the non-modular. We think the reason is somehow improved memory layout for the separately compiled version, which helps the cache system.

Results for lists

nlenrev For the *nlenrev* and the *rlenrev* benchmarks, Kafka is the slowest implementation. The reason is the slow garbage collector, and we have run these benchmarks again with enough heap so that the programs do not have to perform any garbage collection during the program run.[1] The C version has been added to the *nogc* results, because it does not free the memory allocated. The bad performance of the C program is probably due to an inefficient memory manager: we used the C library function `malloc()` for allocating list cells. For Java, we have used the class `LinkedList` from the standard library and used automatic boxing conversions to put integers and floating-point values into lists. As can be seen with these results, boxing and unboxing in Java affects performance very much. The functional implementations, for which boxing is a very common problem, optimize it much better. Opal performs very well on the list benchmarks, because the lists are used in a single-threaded way, and Opal contains special code to dynamically reuse single-threaded heap objects. This reduces memory allocation significantly and thus results in fast execution. Without garbage collection, Kafka performs quite well.

rlenrev The results for *rlenrev* are similar to those for *nlenrev*, with a few exceptions. All implementations are slower, because more memory is allocated. Opal performs worse because of inefficient handling of floating-point values. For the *nogc* version, Kafka is better than in the integer list benchmark, because the specialized representation of list cells containing floating-point values. Better floating-point code when compared to other implementations could also have a small effect.

Results for trees

tree For the *tree* benchmark, we had to increase the heap size for the Kafka measurements, because the created tree is larger than the default heap size. Given enough room, Kafka performs quite well. We can draw the conclusion that data structure handling also performs quite well in our implementation.

8.5 The Pseudoknot Benchmark

The Pseudoknot benchmark is a realistic program from the field of molecular biology. It has been used in an experiment where 25 language implementations have been

[1]The machine on which the tests were performed has not enough physical memory to avoid garbage collection for Java, but we have been able to reduce garbage collection to two invocations.

tested on the same problem (Hartel et al., 1996). The focus of this experiment was on compile and execution times for all implementations. The program calculates the three-dimensional structure of a molecule, based on several constraints on its atoms. It requires a lot of operations on three-dimensional points and uses backtracking search to find all solutions. The result of the program is the distance of the most distant atom of all solutions, which has to be printed with six digits accuracy.

We have ported the Haskell version of the Pseudoknot benchmark to Kafka, which resulted in a program of 3404 lines of code. This is by far the largest program which has been written in Kafka and been tested using our prototype.

For this benchmark, we had to adapt the measuring techniques, because some parts of our prototype system are not mature enough. In particular, we run the benchmark in our prototype without optimization, because the optimizer is too slow and the run time of the actual program too short to give meaningful results. Additionally, we ran the benchmark with 100 MB of heap space in order to reduce invocations of our rather slow garbage collector. We think this is fair, because the C version has no garbage collector at all and the garbage collectors of the other implementations are highly tuned.

We have compared our system to the C, Haskell, ML and Opal versions of the Pseudoknot benchmark. The C and ML versions work with 64-bit floating point numbers, the original Haskell and Opal versions with 32-bit floating-point numbers (this was allowed in the original benchmark, too). We have additionally measured the performance of 64-bit floating-point Haskell and Opal versions in order to compare them to the Kafka version. The C, Haskell and Opal version worked without modification. The ML version had to be slightly modified because the original was written for the Standard ML of New Jersey (SML/NJ) (Appel and MacQueen, 1987) compiler and we were using the MLton compiler. Only changes caused by different ML libraries were required.

Since one reason for a dynamically compiling implementation is to achieve the kind of flexibility offered by an interpreter, we have also measured the run and compile times for running the Pseudoknot program in the GHC interpreter `ghci`. This is a non-optimizing bytecode interpreter, which otherwise shares its implementation with GHC.

We measured the following setups:

– The run time of the C, Haskell, ML and Opal versions, with full optimization switched on.

– The run time of the C and Haskell version, without optimization.

– The run time of the Kafka version (which includes parsing, type checking, conversions and code generation).

– The run time of the interpreted Haskell version.

– The compile times for the C, Haskell, ML and Opal version with optimization.

– The compile times for the C and Haskell version without optimization.

Impl	Run (s)	Comp (s)	Run+Comp (s)	Size (bytes)
C opt	0.035	6.649	6.684	154424
C	0.047	1.197	1.244	126296
Hs opt	0.282	7.837	8.119	460028
Hs	0.691	9.744	10.435	926396
Hs int	⋆ 5.912	⋆ 4.536	10.448	—
Hs opt 64	0.294	7.824	8.118	476860
Hs 64	0.719	9.119	9.838	953180
Hs int 64	⋆ 6.113	⋆ 4.559	10.672	—
ML	0.061	8.410	8.471	276268
Opal	0.174	113.554	113.728	782400
Opal 64	0.760	39.616	40.376	491392
Kafka	⋆ 3.586	⋆ 3.366	6.952	490474

⋆ derived from Run+Comp (see text)

Table 8.11: Pseudoknot results

– The amount of generated code for each configuration (except for the interpreted Haskell version).

– The Haskell and Opal versions were tested both with 32 and 64-bit floating-point numbers

The optimized C version was compiled with optimization option `-O4`, the Haskell version with option `-O2`. The ML version was compiled without compiler options. The Opal versions were compiled with the options `opt=full debug=no`. The Kafka version was run with options `-O0` and `-H100M`.

Table 8.11 gives the results, some of which are graphically shown in Figures 8.1 and 8.2. In the first column, the name of the measured configuration is shown. "C opt" is optimized C code, "C" is unoptimized C code, similar for Haskell (code "Hs"). "ML" is the MLton version. "Opal" is the 32-bit floating-point version. All 64-bit floating-point versions for Haskell and Opal are indicated with "64". "Run" is the run time in seconds reported by the `time` command, "Comp" is the compile time in seconds for compiling the configuration. "Run+Comp" is simply the sum of run and compile times. In the last column, the amount of generated code is shown. For the C, Haskell, Opal and ML version, this is the size of the executable file without any debugging information and symbols (as produced by `strip`), for Kafka, it is the precise amount of machine code. For interpreted Haskell, no machine code is generated, and the size of the generated byte code cannot be determined, therefore, the last column is empty in these cases.

Since compile and run time are difficult to separate in the Kafka system, where compilation and execution is interleaved, we have made the following approximation: We have measured the complete run time of the program using the time stamp counter register of the Pentium processor. This gives the precise number of machine

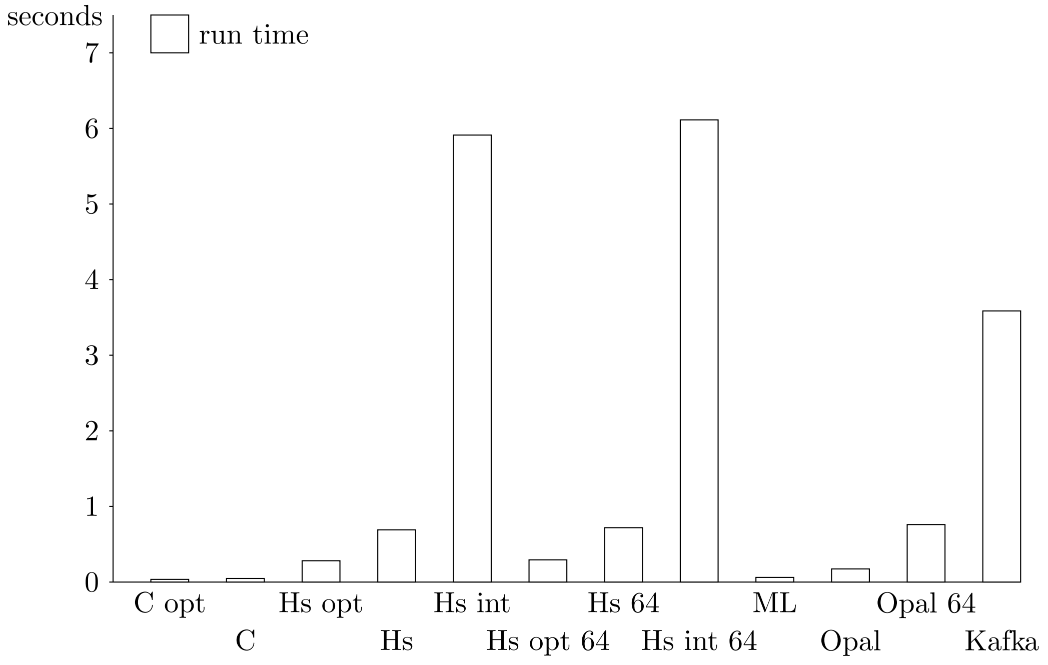

Figure 8.1: Pseudoknot: run times

cycles elapsed. Additionally, we measured the time spent in the code generator using the same technique (excluding the time spent in the context switch to the Haskell environment, but this is hard to measure). The difference of complete time and code generation time is called “useful time”, and includes both the running program and the garbage collector. The percentage of useful time has then been applied to the run time of the program as measured with the `time` command, resulting in the values in Table 8.11. For interpreted Haskell, we have a similar problem: we cannot measure the precise compile and run times. The numbers in the table result from loading the Haskell program in the interpreter and immediately running it (command line `time ghci -XCPP Nuc.hs -e "main"`) and loading it and immediately quitting the interpreter (command line `time ghci -XCPP Nuc.hs -e ":q"`). The run time reported is then the difference between both times.

We have visualized the run times in Figure 8.1 and the combined run and compile times are shown in figure 8.2. The bars for Opal in Figure 8.2 have been clipped and annotated with the combined run and compile time because they are much larger than for the other implementations.

Even though the Kafka implementation is much slower than the other compiled versions, this benchmark proves that efficient code generation is possible for high-level languages at run time. Code is generated at a rate of approximately 142 KB per second. Since our code generator is written in Haskell and not tuned for speed, we think that is a satisfactory result. Our implementation produces code which is an order of magnitude slower, but the run time and the compile time together are lower

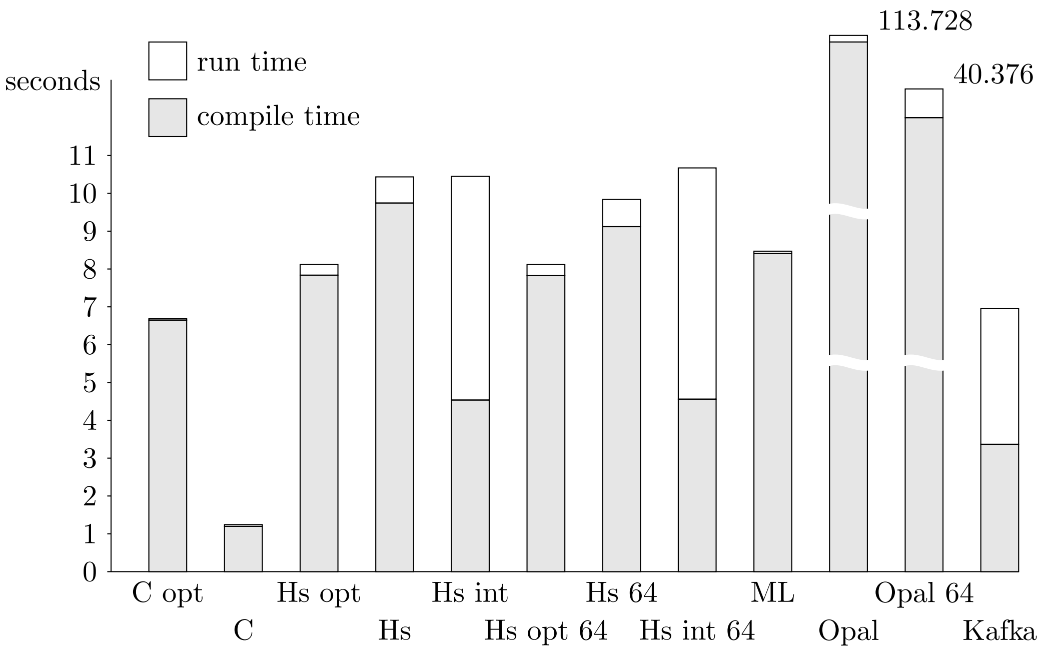

Figure 8.2: Pseudoknot: run and compile times combined

than the compile times of the other implementations, except for the unoptimized C version. The amount of code generated is similar to that produced by the Glasgow Haskell and Opal compilers.

The Opal versions are very fast, where the 64-bit floating-point version is slower than the 32-bit version. This is similar to the results for the *rfib/ffib* benchmarks. For Haskell, there is not much difference for both floating-point sizes. The Opal compilation times are very large, but note that much of the compilation time is spent in the C compiler, since Opal compiles to C. Interestingly, the compile time for the 64-bit floating-point version and the code size are lower than the 32-bit version, even though it is slower. The reasons for this anomaly are unclear, maybe there is some difference in inlining opportunities which could explain this. The compile times for MLton are not much larger than for GHC. This is probably due to the fact that the Pseudoknot program consist of very large static data structures, which MLton seems to handle efficiently.

The Kafka version runs by a factor of about 10 slower than the compiled Haskell or Opal version. We think that careful tuning of the optimizer, so that it can run on programs of this size, will reduce this factor so that results similar to the ones of the small benchmarks presented in the previous section can be achieved. Compared to interpreted Haskell, our implementation is much faster, even though the incremental dynamic compilation method provides in principle the same degree of flexibility.

Line	Incremental	Input	Tot.Run (MCy)		Codegen (MCy)		# Gen	Size
1	No	0	7868		3642		1	495082
2	No	1	10321		4079		11	508764
3	Yes	0	5192	⋆0.66	868	⋆0.23	3	811
4	Yes	1	9530	⋆⋆0.92	3158	⋆⋆0.77	134	391034
with profiling								
5	Yes (1st)	0	5104	⋆0.65	930	⋆0.26	3	881
6	Yes (2nd)	0	6122	⋆0.78	713	⋆0.20	2	780
7	Yes (1st)	1	11217	⋆⋆1.09	3293	⋆⋆0.81	131	413485
8	Yes (2nd)	1	12164	⋆⋆1.18	4456	⋆⋆1.09	12	534867

⋆ relative to line 1, ⋆⋆ relative to line 2

Table 8.12: Results for incremental compilation

8.6 Effects of Incremental Compilation

The incremental compilation feature described in Chapter 5 affects run time in two directions: first, compilation time is reduced because less code needs to be generated. Second, when all code in the program is executed, compilation time is increased because the placement of delay expressions, the generation of calls to the code generator, the invocation of the generation, patching code and accessing free variables from the stack induce run time overheads.

In order to quantify these effects, and to see whether the technique of incremental compilation we have developed has any advantages, we have performed another experiment.

We have taken the Pseudoknot benchmark described in the previous section and have modified it so that the complete program is contained in a function. Additionally, the main program asks the user for a number: when a 1 is given, the benchmark is run, otherwise it is not run.

Because loading, type checking and converting takes so long with our prototype implementation, we report different numbers in this section than the ones in the previous section. For all program runs, we report the time spent in execution and code generation as measured by the Kafka virtual machine using the `rdtsc` instruction. All times reported in this section are measured in million machine cycles (megacycles ≡ MCy). The time needed for parsing, type checking, CPS- and closure-converting are not included in the reported numbers. Optimizations were turned off again, as in the previous section.

Results for Incremental Compilation

The results are shown in Table 8.12. The first column contains the line number to be referred to later, column two tells whether the Pseudoknot program was executed with incremental compilation switched on ("Yes") or off ("No"). The third column gives the input to the program (1: run the benchmark, 0: do not run the benchmark). The fourth column gives the number of megacycles spent executing the program (this includes code generation and garbage collection time). The fifth column report the number of megacycles spent in the code generator, and the sixth column tells how often the code generator was invoked. The last column gives the amount of generated code. The second numbers in the columns labelled "Tot.Run" and "Codegen" for the incrementally compiled versions are the total run times or code generation times, respectively, relative to the numbers in line 1 (for input 0) and 2 (for input 1).

Lines 1 and 2 are the results when incremental compilation is switched off. Lines 3 and 4 show them when incremental compilation is enabled, but when profiling feedback is switched off, and lines 5 to 8 give the numbers when both features are enabled. The lines labelled "1st" correspond to a run where no previous profiling information is available, the lines labelled "2nd" give the numbers when a previous profiled run has already completed and profiling data is available.

When the input to the benchmark program is 0, that is, when the expensive backtracking search is not invoked, execution time is of course lower than when the search is performed. In the case that incremental compilation is switched off (lines 1 and 2), running the complete program (input 1) causes 11 code generations and about 10 KB more code to be generated than for input 0. This is due to the fact that some functions are polymorphic and are thus compiled when they are first called even when incremental compilation is not enabled.

Line 3 shows that incremental compilation can reduce the amount of code which is generated dramatically (of course, our benchmark is artificial, for real programs the differences are probably less pronounced). Even though the number of code generator invocation increases, the time spent in code generation and in total run time is also reduced: code generation time is reduced by 77%, run time by 34%.

When the complete program is executed and incremental compilation is enabled (line 4), we also see that the amount of code, the code generation time and the total run time is less than for the non-incrementally compiled version. Even though code generation is performed more than 10 times as often, total run time is reduced by 8%. This effect is due to the fact that code generation is performed more quickly. Our investigation of this astonishing fact revealed that this is probably an artifact of our implementation: the placement of delay expression in the syntax tree speeds up some analysis in the code generator. For example, the code which calculates how much heap space is allocated in a function stops as soon as a delay expression is encountered, because the deferred compilation of the body expressions will have their own heap check instructions emitted when they are generated later. This means that for more efficient or different analysis and code generation techniques, the incremental version will probably not be faster than the non-incremental, because the additional code which is generated for invoking the code generator will slow down the programs.

In lines 5–8 we can see that our heuristic of not delaying functions which have been used in the last program run indeed reduces the number of delay expressions in the syntax tree and thus reduces the invocations of the code generator. The amount of code generated increases because instrumentation code is generated to count how often functions/continuations and branch alternatives are executed. Additionally, the algorithm for placing delay expressions needs to consult the tables of the previous run in order to decide where to place them. This also costs time. But note that in line 6, we can see that code generation time is reduced when compared to line 3, where no profiling information is used. This encourages more research of the practicability of profiling-based feedback in our setting.

8.7 Discussion

There are several interesting facts to be discussed:

– The Kafka implementation is able to reach a factor of two of the run times of very mature and efficient language implementations for Standard ML, Haskell and Scheme. For some benchmarks, Kafka is even faster than other implementations.

– The garbage collector currently implemented in our prototype is very slow. So it is to be expected that the performance of our prototype underestimates the performance of our approach in general.

– Modular programming in Haskell in the form of laziness, type classes and separate compilation has its price. Our approach could be used to overcome the performance problems due to abstraction and type classes. Whether it is also suitable for lazy languages remains an open research problem.

– The run times for Java prove that dynamically compiling system can offer very good performance.

– The very good performance of the programs compiled by MLton can be seen as an upper bound of our approach: MLton performs specialization and defunctionalization, so the code actually executed is monomorphic as in our case. The drawback of MLton is the restriction to whole-program compilation and slower compilation. Our approach could again be a solution, but a more complete implementation is needed to prove that expectation.

– When comparing the Pseudoknot results for interpreted Haskell and Kafka, we draw the conclusion that it is possible to implement a functional language so that it is as flexible as an interpreter, but with performance close to a statically compiled implementation.

– Incremental compilation has the ability to reduce both code generation time and overall run time for dynamically compiled functional programs. The usefulness of profiling-based feedback needs further investigation.

In conclusion, the results presented in this chapter prove that it is possible to efficiently produce high-quality machine code for a high-level language at run time.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we have presented several techniques for the implementation of a run-time compiler for functional programming languages. Our specialization methods have the potential to eliminate the overheads of many language constructs heavily used in modern functional programming languages: polymorphic functions, polymorphic data types and type classes, as well as proposed extensions: open functions and data types, and parameterized modules. The ideas have been implemented in a prototype system, which can perform monomorphization of polymorphic functions and data types and additionally carries out a number of standard optimizations, of which some had to be modified to play nicely with the dynamic compilation approach. The basic idea of using run-time specialization for the implementation of functional languages looks promising: our experiments show that both generation of efficient code and fast compile times are possible for functional programming languages.
With respect to the problems identified in Section 1.1.1, we give the following answers:

Feasibility of dynamic compilation for functional programs We have shown that dynamic compilation of functional languages can be performed efficiently. The compilation times reported in Chapter 8 are already encouraging, and we expect that with some engineering effort they can be reduced further.

Balance of compilation and execution time The typed dynamic CPS representation we have developed enables fast code generation as well as sophisticated optimizations, both prior to execution and at run-time. The low-level nature of closure-converted CPS terms allows efficient code generation, while also providing a well-studied semantics and a wide variety of proven optimization techniques.

Suitability of classic optimizations We have implemented several optimizations known from static functional compilers and have measured their effect on compile and run time.

Techniques from other dynamic language implementations We have taken the inline caching optimization from the implementation of object-oriented programs and have shown its usefulness for functional languages. The incremental compilation and delayed compilation of polymorphic functions is reminiscent of lazy method compilation on object-oriented virtual machines.

Influence on dynamic language constructs We have shown how our specialization technique can eliminate the overheads of parametric polymorphic functions and data types. We have also shown that ad-hoc polymorphism in the form overloaded operators and literals is supported as well. The extension to other, more dynamic language features should be relatively straightforward (see also below in Section 9.2.2). This success encourages experimentation with other dynamic language features.

Summary of Contributions

We have made contributions in the following fields:

Compiler Design We have designed a typed continuation-passing style intermediate representation, which is well suited for type-based dynamic optimizations. The language supports many language features required for a modern functional language, such as type inference, polymorphic functions and algebraic data types.

Optimization Techniques We have designed two techniques for dynamic specialization: first, a monomorphization algorithm for polymorphic functions and overloaded operators. Second, a data type representation and layout analysis, which is also based on dynamic specialization.

Language Implementation The techniques have been implemented in a working prototype, which is able to run programs of realistic size. The implementation is highly configurable: all optimizations can be switched on or off independently and various parameters of the optimizer and code generator can be dynamically modified. This gives a good test bed for experimenting with optimizations and heuristics.

Experimental Results The performance of the generated code as well as the time consumed by the dynamic compiler have been quantified. We have also compared the performance against other functional language implementations, none of which supports dynamic code generation comparable to ours.

9.2 Future Work

We see possibilities for future work both in theoretical and in practical directions.

9.2.1 Theoretic Model

A complete theoretical model, possibly in some kind of specialization calculus, could improve the understanding of the dynamic interaction in a dynamically specializing system, especially in the presence of dynamic optimization. The field of partial evaluation could be a good starting point.

9.2.2 Open Data Types and Open Functions

Functional programming languages normally impose a closed-world assumption: all parts of a program are available at compile time, at least in the form of precompiled libraries or interface files. This view precludes usage of modern techniques like plug-ins or extensions, which can be loaded into a program when requested. Object-oriented languages feature inheritance and subtyping for compiling and type checking separately developed program parts (classes and interfaces), and therefore support the development of such plug-ins. Algebraic data types, on the other hand, do not support extension, and are therefore less suited for developing dynamic applications. Type classes as in Haskell are open, but are less suited for modeling many problems for which algebraic data types are normally used. As a solution, *open data types* and *open functions* have been proposed. Löh and Hinze (2006) describe a language with open data types and open functions and give a translation into normal Haskell, but under the above-mentioned closed-world assumption. We could make use of our existing machinery for run-time compilation in order to allow extension of both algebraic data types and functions at run-time.
As an example, consider an algebraic data type in Haskell, which represents the abstract syntax of lambda calculus terms:

```
data E = Var String | App E E | Abs String E
```

and the fragments of an evaluator for these terms (the syntax `open eval` declares `eval` as an open function):

```
open eval
eval (Var x) = ...
eval (App e1 e2) = ...
eval (Abs s e) = ...
```

In the open data type and open function approach, it is possible to define additional constructors for the data type and additional cases for the function, for example:

```
data E = ... | Const Int
eval (Const i) = ...
```

Open data types and open functions fit nicely into our framework: open data types dynamically extend the defined constants (both at type and term level), and open functions simply extend given function definitions. For this to work, imagine an input

language which provides the definition of functions by equations. This language can be translated to a language with case statements easily (Wadler, 1987).
For the evaluator example above, the first data declaration and the first set of function definition gives the following program:

```
data E = Var String
       | App E E
       | Abs String E
eval (Var x) = ...
eval (App e1 e2) = ...
eval (Abs s e) = ...
```

When the second set of definitions is loaded, the program is extended to the following:

```
data E = Var String
       | App E E
       | Abs String E
       | Const Int
eval (Var x) = ...
eval (App e1 e2) = ...
eval (Abs s e) = ...
eval (Const i) = ...
```

The dynamic specialization mechanism will automatically use the new set of definitions for the rest of the program evaluation.
When dynamically extending a set of functions, it is of course necessary to ensure the correct ordering of functions. Again, a sorting algorithm which takes a certain order relation on function definitions into account can be used.
A necessary extension of our dynamic compiler would be the possibility to invalidate compiled code when new program fragments are loaded, as these may render earlier compilations incorrect.

9.2.3 Type Classes

In our framework, type classes and instances could be implemented by adding type-specific case expressions (or in the extension from the previous section: type-specific function equations) to the syntax. Returning to our example of the type class `Eq` from Chapter 6, we would simply add specific equality functions for all types which should be instances:

```
eq (Int: *) (x: Int) (y: Int) = intEq x y
eq (Bool: *) (True: Bool) (True: Bool) = True
eq (Bool: *) (False: Bool) (False: Bool) = True
eq (Bool: *) (x: Bool) (y: Bool) = False
```

Note how the more specific equations for the equality function for `Bool` come first, and the last equation (using variables as parameters) denotes the default case.
The default method declarations in Haskell type classes, which allow to define methods in the body of a class declaration, can also be defined in our system. We can add the following definition to our functions:

```
neq (a: *) (x: a) (y: a) = not (eq a x y)
```

Since this function will be automatically specialized for all types for which it is used, the effect is the same as for default methods.
Jones (1995) has also investigated the effect of specialization for implementing type classes. In contrast to the usual dictionary-passing implementations, Jones specializes programs at compile-time and resolves all applications of overloaded functions. This technique also relies on a closed-world assumption, and our system results in the same degree of specialization, but in an open dynamic framework. Interestingly, Jones reports that the specialization of type classes actually *reduces* the amount of binary code, instead of the expected code growth. This is a very encouraging result.

9.2.4 Better Compiler

The current system design has been implemented as a working prototype which can execute small to medium-sized programs efficiently. The logical next step in the development is to complete the implementation and to make it more robust and portable. The addition of more optimizations should be possible, but might require more information about the program to be derived by additional analyzers. Several important issues for a practical system have not yet been implemented. Proper code buffer management is not yet realized in the prototype.
The applicability of the technique presented in this thesis to language abstractions present in the ML family of languages, e.g. functors, has also not yet been under investigation.
The design and implementation of specialization strategies, as well as an empirical study of their effectiveness should be carried out. Another interesting avenue for future work is the automatic (for example profiling-based) generation of application-specific strategies.
The runtime compiler and the runtime system could and should be improved in several ways. It needs a better register allocator, a better garbage collector etc.
Several specific topics for improvement are the following:

Overflow Checking A reliable programming language should support checks on overflow for arithmetic operations. Standard ML and Opal, for example, check all arithmetic operations and raise exceptions (or abort the program) if an overflow happens. The MLton compiler shows that overflow checking does not necessarily cause a significant slowdown for arithmetic-intensive programs, as can be seen from the results in Chapter 8.

Type System The type system of the current implementation is very basic. It should be extended to type classes and probably other modern type system features which have proven useful in practice.

Improved Data Representation Our data representation does not yet exploit all possibilities of type based representation analysis. For example, algebraic data types which have only one variant with one field are "transparent", and directly use the representation of the field. When a data type has only one variant with fields (which means, only one boxed variant), a constructor tag could be omitted. This saves memory and speeds up case distinctions because it removes a redundant test.
Pointers to heap cells could also be tagged with special values in the low bits in order to speed up case expressions. For data types with few boxed variants, the constructor tag could be stored in unused bits in the pointer instead of bits in the header word of the heap-allocated record. The use of tagged pointers has a performance benefit on modern machines which normally have relatively precise branch prediction and high clock rates (Marlow et al., 2007).
Additionally record fields which are used together should be placed near to each other. This arrangement either needs profiling data to determine the "hottest" fields or some heuristic which captures the real behavior of the system. In addition, it may be useful to pack fields in such a way that unboxed values are separate from pointers, because this simplifies the descriptors needed for garbage collection. Currently, our implementation uses bitmaps to indicate which words of a record may contain pointers.
Other useful data representation optimizations include splitting arrays of pairs into pairs of arrays. This also increases locality and avoids gaps due to alignment restrictions.

Graph-based Intermediate Language The prototype is based on tree-shaped intermediate languages, which naturally arise from the use of algebraic data types in functional languages. Unfortunately, it turns out that several transformations are quite slow (for example dead variable elimination), because most of the intermediate program must be copied even if relatively few modifications to the tree happen. An alternative is to use a graph-based intermediate representation, such as the one used by Kennedy (2007). The program graph contains links between variable definitions and occurrences, so that substituting a variable for another can be performed in constant time, and operations such as finding all uses of a variable is very efficient. Another alternative would be to use a zipper (Huet, 1997) data structure, in the form of control-flow graphs based on a zipper-like structure (Ramsey and Dias, 2005).

9.2.5 Instrumentation and Recompilation

Just-in-time compiling virtual machines for Java and other object-oriented programming languages support instrumentation and recompilation. When a program is executed, the bytecode is first either interpreted or compiled using a simple, but

fast compiler. The interpreter or the code generated by the simple compiler counts events, such as method invocations. When these counters indicate (by some heuristic measure) that an event has occurred often enough, the code for the method is either compiled (for the interpreter) or recompiled at a higher optimization level (for the compiled version). The goal is to adapt the generated code to the dynamic behavior of the program under execution.
Integration of a recompilation mechanism into our prototype would be desirable, as it is necessary for advanced research in dynamic compilation and optimization. We have implemented a basic instrumentation infrastructure, but it is currently only used for controlling the placement of delay expressions when incremental compilation is enabled.

9.2.6 Value Specialization

It would be possible to reuse the specialization mechanism for value specialization by changing the heuristics which inserts calls to the run-time specializer. Currently, the specializer is only called for type parameters in order to remove polymorphism, but changing that would allow the code to be specialized for run-time parameters such as the underlying system or tuning parameters. Similarly to the approach taken in the Fabius system by Leone and Lee (1998) one could implement a sort of staging for functions, based on currying; or we could request annotations from the programmer, as in staged languages such as MetaML (Taha and Sheard, 2000). Whaley (1999) has investigated value specialization for Java.

List of Figures

List of Tables

Bibliography

A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 280–290. ACM Press, 1998.

A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1), Feb. 2003.

J. Agat. Types for register allocation. In *Proceedings of IFL'97*, volume 1467 of *Lecture Notes in Computer Science*, 1997.

O. Agesen. The Design and Implementation of Pep, A Java^{TM} Just-In-Time Translator. *Theory and Practice of Object Systems*, 3(2):127–155, 1997.

A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

A. W. Appel. Loop Headers in Lambda-Calculus or CPS. Technical Report TR-460-94, Princeton University, 1994.

A. W. Appel. SSA is Functional Programming. *ACM SIGPLAN Notices*, 33(4): 17–20, 1998.

A. W. Appel and D. B. MacQueen. A Standard ML Compiler. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Research Report 23429, IBM Research, Nov. 2004.

M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, Feb. 2005.

L. Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 65–73, New York, NY, USA, 1993. ACM Press.

J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, Aug. 1978.

J. Backus, R. Beeber, S. Best, R. Goldberg, L. Haibt, H. Herrick, R. Nelson, D. Sayre, P. Sheridan, H. Stern, I. Ziller, R. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings Western Joint Computer Conference*, pages 188–198, Los Angeles, California, Feb. 1957.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

V. Balat and O. Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In *TIC '98: Proceedings of the Second International Workshop on Types in Compilation*, pages 240–252, London, UK, 1998. Springer-Verlag.

L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium$^{\text{TM}}$-based systems. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 191, Washington, DC, USA, 2003. IEEE Computer Society.

H. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science, (Background: Computational Structures)*, volume 2. Oxford University Press, 1992.

J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating (extended abstract). In *Proceedings of Workshop on Unexpected Software Evolution (USE'03)*, Apr. 2003.

Bigloo Developers. Bigloo Homepage. Available from: `http://www-sop.inria.fr/mimosa/fp/Bigloo/`, 2008. Last visited: 2008-10-07.

R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, New York, NY, USA, 2006. ACM Press.

U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, Gothenburg, Apr. 1999.

G. Bray. Implementation implications of Ada generics. *Ada Letters*, III(2):62–71, 1983.

P. J. Brown. Throw-away compiling. *Software Practice and Experience*, 6(4):423–434, 1976.

D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275. IEEE Computer Society, 2003.

D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, Sept. 2004.

T. Brus, M. C. J. D. van Eekelen, M. van Leer, M. J. Plasmeijer, and H. Barendregt. Clean - a language for functional graph rewriting. In Kahn, editor, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA*, number 274 in Lecture Notes in Computer Science, pages 364–384. Springer-Verlag, 1987.

R. G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme.* PhD thesis, Indiana University Computer Science Department, Mar. 1997.

R. G. Burger and R. K. Dybvig. An infrastructure for profile-driven dynamic recompilation. In *IEEE Computer Society 1998 International Conference on Computer Languages*, May 1998.

M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM Press.

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, Stanford University, Stanford, California, Aug. 1980. ACM Press.

H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming*, pages 56–71, Mar. 2000.

G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105, New York, NY, USA, 1982. ACM Press.

C. Chambers and D. Ungar. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 146–160. ACM Press, 1989.

C. J. Cheney. A non-recursive list compaction algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.

A. Chernoff and R. Hookway. Digital FX!32 – Running 32-Bit x86 Applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop*, Seattle, Washington, Aug. 1997.

C. Consel and O. Danvy. Partial evaluation: Principles and perspectives. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 1993.

K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, New York, NY, USA, 1998. ACM Press.

R. J. Dakin and P. C. Poole. A mixed code approach. *Computer Journal*, 16(3): 219–222, 1973.

L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings 9'th ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, N.M, Jan. 1982. ACM Press.

O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

J. L. Dawson. Combining interpretive code with machine code. *Computer Journal*, 16(3):216–219, 1973.

D. Detlefs and O. Agesen. Inlining of virtual methods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag, 1999.

L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302. ACM Press, 1984.

R. B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, 1975.

R. Dockins and S. Z. Guyer. Bytecode verification for Haskell. Technical report, Tufts University, Feb. 2007.

DotGNU Project. DotGNU Project/Portable.NET. Available from: `http://dotgnu.org`, 2004. Last visited: 2008-10-07.

D. Dubé. *Demand-Driven Type Analysis for Dynamically-Typed Functional Languages.* PhD thesis, Université de Montréal, Aug. 2002.

K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. Technical Report 8502, IBM, 1996.

ECMA International. *Standard ECMA-334: Common Language Infrastructure (CLI).* 3rd edition, June 2005.

A. Ershov. On the partial computation principle. *Information Processing Letters*, 6 (2):38–41, 1977.

M. Feeley and G. Lapalme. Closure generation based on viewing lambda as epsilon plus compile. *Journal of Computer Languages*, 17(4), 1992.

A. Fischbach and J. Hannan. Type systems for useless-variable elimination. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 25–38, London, UK, 2001. Springer-Verlag.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

M. S. O. Franz. *Code-Generation On-the-fly: A Key to Portable Software.* PhD thesis, ETH Zürich, 1994.

C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

GCC Developers. GNU Compiler Collection Homepage. Available from: `http://gcc.gnu.org`, 2008. Last visited: 2008-10-07.

GHC Developers. Glasgow Haskell Compiler Homepage. Available from: `http://www.haskell.org/ghc`, 2008. Last visited: 2008-10-07.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification.* Addison-Wesley, 2nd edition, June 2000.

M. Grabmüller. Implementing Closures using Run-time Code Generation. Research report 2006-02 in *Forschungsberichte Fakultät IV – Elektrotechnik und Informatik*, Technische Universität Berlin, Feb. 2006.

M. Grabmüller. A model of functional programming with dynamic compilation and optimization. In H. Nilsson, editor, *Trends in Functional Programming*, volume 7. Intellect, Apr. 2007.

M. Grabmüller. rdtsc – binding for the rdtsc machine instruction. Available on the World Wide Web: `http://uebb.cs.tu-berlin.de/~magr/projects/rdtsc/doc/`, June 2008. Last visited: 2008-10-07.

M. Grabmüller and D. Kleeblatt. Harpy: Run-time code generation in Haskell. In *Haskell Workshop 2007*. ACM Press, Sept. 2007.

B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248 (1–2):147–199, Oct. 2000.

D. Gries and N. Gehani. Some ideas on data types in high-level languages. *Communications of the ACM*, 20(6):414–420, 1977.

L.-J. Guillemette and S. Monnier. A type-preserving closure conversion in Haskell. In *Haskell Workshop 2007*, 2007.

C. V. Hall, K. Hammond, S. L. P. Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, Mar. 1996.

J. Hannan. Type systems for closure conversions. In H. R. Nielson and K. L. Solberg, editors, *Proceedings of Workshop on Types for Program Analysis*, number PB-493 in Daimi Reports, pages 48–62, 1995.

J. Hannan and P. Hicks. Higher-order arity raising. In *Proceedings of 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, Baltimore, MD, Sept. 1998.

J. Hannan and P. Hicks. Higher-order uncurrying. *Journal of Higher Order and Symbolic Computation*, 13(3):179–216, 2000.

R. Harper and M. Lillibridge. Polymorphic type assignment and CPS conversion. *Lisp and Symbolic Computation*, 6(3–4):361–379, Nov. 1993.

R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, New York, NY, USA, 1995. ACM Press.

P. H. Hartel, M. Feeley, M. Alt, L. Augustson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas,

P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with “pseudoknot”, a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.

K. Hazelwood Cettei. *Code Cache Management in Dynamic Optimization Systems*. PhD thesis, Harvard University, May 2004.

M. Hicks. *Dynamic Software Updating*. PhD thesis, Computer and Information Science Department, the University of Pennsylvania, Aug. 2001.

U. Hölzle. *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Computer Science Department, Stanford University, 1994.

C. Hope and G. Hutton. Accurate step counting. In *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages*, Dublin, Ireland, 2005.

L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–375, Dec. 1999.

P. Hudak, J. Peterson, and J. H. Fasel. A gentle introduction to Haskell. World Wide Web, 1999.

G. Huet. Function Pearl: The Zipper. *Journal of Functional Programming*, 7(5): 549–554, Sept. 1997.

Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, June 2005. Available from: `http://www.intel.com/design/Pentium4/documentation.htm`.

Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volumes 1, 2A, 2B, 3A, 3B*, Jan. 2006. Available from: `http://www.intel.com/design/Pentium4/documentation.htm`.

R. L. Johnston. The dynamic incremental compiler of APL\3000. In *APL '79: Proceedings of the International Conference on APL: part 1*, pages 82–87, New York, NY, USA, 1979. ACM Press.

M. P. Jones. The implementation of the Gofer functional programming system. Research report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994.

M. P. Jones. Dictionary-free overloading by partial evaluation. In *Proceedings of PEPM'95*, 1995.

S. L. P. Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

R. Kelsey, W. Clinger, J. Rees, et al. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(6):26–76, Sept. 1998.

R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. *SIGPLAN Notices*, 30(3):13–22, 1993.

A. Kennedy. Compiling with continuations, continued. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 177–190, New York, NY, USA, 2007. ACM.

B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 2nd edition, 1988.

A. Klaiber. The technology behind Crusoe processors. Available from: `http://www.charmed.com/PDF/CrusoeTechnologyWhitePaper_1-19-00.pdf`, 2000. Last visited: 2008-10-07.

N. Kobayashi. Type-based useless-variable elimination. *Higher-Order and Symbolic Computation*, 14(2-3):221–260, 2001.

T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 111–120, New York, NY, USA, 2005. ACM Press.

O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

D. Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Utrecht University, Nov. 2003.

D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Indiana University Computer Science Department, Sept. 1997.

M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, June 1994.

M. Leone and P. Lee. A declarative appoach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, 1996.

M. Leone and P. Lee. Dynamic specialization in the fabius system. *ACM Computing Surveys*, 30(3es):23, 1998.

X. Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, New York, NY, USA, 1992. ACM Press.

X. Leroy. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation*, Amsterdam, The Netherlands, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Sun Microsystems, 2nd edition, 1999.

A. Löh and R. Hinze. Open data types and open functions. In M. Maher, editor, *Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, Venice, Italy, July 2006. ACM Press.

D. Lomov and A. Moscal. Dynamic Caml – Run-Time Code Generation Library for Objective Caml. Available on the World Wide Web at `http://oops.tepkom.ru/dml/`, last visited: 2008-10–7, May 2002.

D. Luna, M. Pettersson, and K. Sagonas. Efficiently compiling a functional language on amd64: The hipe experience. In *7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP 2005)*, 2005.

S. Marlow, A. R. Yakushev, and S. Peyton Jones. Faster laziness using dynamic pointer tagging. In *ICFP 2007*, 2007.

H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

C. S. McDonald. fsh – a functional unix command interpreter. *Software Practice and Experience*, 17(10):685–700, Oct. 1983.

J. Meacham. jhc. Available from: `http://repetae.net/john/computer/jhc/`, 2007. Last visited: 2008-10-07.

G. Meehan and M. Joy. Compiling lazy functional programs to Java bytecode. *Software Practice and Experience*, 29(7):617–645, 1999.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997. Revised edition.

Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *In Proc. 23rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

MLton Developers. MLton Standard ML Compiler Homepage. Available from: `http://mlton.org`, 2006. Last visited: 2008-10-07.

M. Mock. *Automating Selective Dynamic Compilation.* PhD thesis, University of Washington, Aug. 2002.

Mono Project. Mono. Available from: `http://www.mono-project.com`, 2007. Last visited: 2008-10-07.

G. Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, Dec. 1995. Published as CMU Technical Report CMU-CS-95-226.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342–371, 1991.

G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

H.-D. Nguyen and A. Ohori. Record unboxing. World Wide Web: `http://www.pllab.riec.tohoku.ac.jp/~ohori/research/RecordUnboxing.pdf`, May 2007. Last visited: 2008-10-07.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

K. Ogata and N. Doi. Object allocation and dynamic compilation in MultithreadSmalltalk. In *SAC '94: Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 452–456, New York, NY, USA, 1994. ACM Press.

F. Ogel, G. Thomas, and B. Folliot. Supporting efficient dynamic aspects through reflection and dynamic compilation. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1351–1356, New York, NY, USA, 2005. ACM Press.

Opal Group. Opal Project Homepage. Available from: `http://uebb.cs.tu-berlin.de/~opal`, 2004. Last visited: 2008-10-07.

M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} Server Compiler. In *Proceedings of the Java^{TM} Virtual Machine Research and Technology Symposium (JVM '01).* USENIX Association, Apr. 2001.

W. Partain. The nofib benchmark suite of haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

P. Pepper. *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer-Verlag, 2nd edition, 2003.

N. Perry and E. Meijer. Implementing functional languages on object-oriented virtual machines. Microsoft White Paper.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, Apr. 2003. Also available from: `http://www.haskell.org/haskellwiki/Definition`, last visited: 2008-10-07.

S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 636–666, Cambridge, Massachussets, USA, 26–28 August 1991. Springer-Verlag LNCS523.

B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusettes, 2002.

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

C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1 (1):11–32, Winter 1988.

N. Ramsey and J. Dias. An applicative control-flow graph based on huet's zipper. In *ACM SIGPLAN Workshop on ML*, pages 101–122, Sept. 2005.

J. Reppy. Local CPS conversion in a direct-style compiler. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, pages 13–22, 2001.

A. Rossberg. The missing link: dynamic components for ML. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 99–110, New York, NY, USA, 2006. ACM Press.

A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. Alice through the looking glass. In *Trends in Functional Programming*, volume 5, pages 79–96. Intellect, 2006.

D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London, September 1990.

M. Schinz and M. Odersky. Tail call elimination on the Java virtual machine. In N. Benton and A. Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2001.

M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton University, Nov. 1994.

Z. Shao and A. W. Appel. A type-based compiler for standard ml. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 116–129, New York, NY, USA, 1995. ACM Press.

Z. Shao and A. W. Appel. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, Jan. 2000.

O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

O. Shivers. Supporting dynamic languages on the Java virtual machine. Technical Report AIM-1576, MIT AI Laboratory, 1996.

J. M. Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Dec. 1999.

J. E. Smith and R. Nair. *Virtual Machines – Versatile Platforms for Systems and Processes.* Morgan Kaufman, 2005.

M. Sperber and P. Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–225, 1997.

D. Spinellis. An implementation of the Haskell language. Master's thesis, Imperial College, June 1990.

P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.*, 19(1):48–86, 1997.

D. Stewart. hs-plugins – Dynamically Loaded Haskell Modules. World Wide Web: `http://www.cse.unsw.edu.au/~dons/hs-plugins/`, Feb. 2006. Last visited: 2008-10-07.

D. Stewart and M. M. T. Chakravarty. Dynamic applications from the ground up. In *Proceedings of the ACM SIGPLAN Workshop on Haskell.* ACM Press, Sept. 2005.

G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 183–194, New York, NY, USA, 2005. ACM Press.

B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, New York, 1986.

T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

Sun, Inc. Java SE Downloads - Previous Release - J2SE 5.0. Available from: `http://java.sun.com/javase/downloads/index_jdk5.jsp`, 2008. Last visited: 2008-10-07.

W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Nov. 1999.

W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *PEPM*, 1991.

W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, Oct. 2000.

D. Tarditi. *Design and Implementation of Code Optimiziations for a Type-Directed Compiler for Standard ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, Dec. 1996. Available as Technical Report CMU-CS-97-108.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. Til: A type-directed compiler for ml. In *ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 181–192, New York, 1996. ACM Press.

K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

A. Tolmach. Combining closure conversion with closure analysis using algebraic types. In *Types in Compilation workshop*, June 1997.

D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *DYNAMO '00: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 41–51, New York, NY, USA, 2000. ACM Press.

D. Ung and C. Cifuentes. Dynamic binary translation using run-time feedbacks. *Science of Computer Programming*, 60(2):189–204, Apr. 2006.

D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3), 1991.

P. Unnikrishnan, G. Chen, M. Kandemir, and D. R. Mudgett. Dynamic compilation for energy adaptation. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, pages 158–163, New York, NY, USA, 2002. ACM Press.

A. van Weelden and R. Plasmeijer. A functional shell that dynamically combines compiled code. In *Proceedings of the 15th International Workshop on the Implementation of Functional Languages, IFL 2003*, volume 3145 of *Lecture Notes in Computer Science*, Scotland, 2004. Springer-Verlag.

P. Wadler. *The Implementation of Functional Programming Languages*, chapter Efficient compilation of pattern matching, pages 78–103. Prentice-Hall, 1987.

D. Wakeling. The dynamic compilation of lazy functional programs. *Journal of Functional Programming*, 8(1):61–81, Jan. 1998a.

D. Wakeling. Mobile Haskell: Compiling lazy functional programs for the Java virtual machine. In *Proceedings of the 1998 Conference on Programming Languages, Implementations, Logics and Programs (PLILP'98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 335–352, Sept. 1998b.

M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.

D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual.* PTR Prentice Hall, 1994.

J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and modal-ml. In K. D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 224–235, Montreal, Canada, June 1998. ACM Press.

N. Wirth and J. Gutknecht. *Project Oberon – The Design of an Operating System and Compiler.* Addison-Wesley, 1998.

Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.

M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 241–252, New York, NY, USA, 2001. ACM Press.

C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent Execution, no Recompilation. *IEEE Computer*, 33(3):47–52, Mar. 2000.