A Mobile Functional Object Code

Manuel M. T. Chakravarty

Institute of Information Sciences and Electronics
University of Tsukuba, Japan
`chak@is.tsukuba.ac.jp`
`www.score.is.tsukuba.ac.jp/~chak`

Abstract. This paper introduces a monadic object code for functional languages that allows an architecture independent, compact representation of program modules; it includes support for distributed applications by featuring a light-weight thread model and asymmetric communication. It proposes to offset inefficiencies introduced by the architecture independence by specialising and optimising the code before or during execution. The proposal is substantially different from classical byte code in both the structure of the language, as it is a strongly-typed functional language encapsulating side-effects in monadic form, and the representation of the code, as it is a bit-aligned structurally, compressed code designed for runtime code-generation, not for interpretation.

1 Introduction

Network computing requires us to re-think the implementation techniques for functional languages. A network-aware execution environment should run on heterogenous clusters, support multi-threading, allow code migration, and provide a simple, yet powerful communication model. The proposed monadic object code, called FOC, supports these features. It allows an architecture independent and compact representation of program modules including a light-weight thread model and asymmetric communication. Although the architecture independence is important for mobility in a heterogenous environment, it adversely affects execution time. These inefficiencies are minimized by code optimisations before or during execution—employing a mix of standard techniques for runtime code generation and new optimisations based on the functional origin of the code.

FOC is not a classical byte code, but a strongly-typed functional language encapsulating side-effects by representing computations in monadic form. Its structure and declarative semantics simplify optimisations during specialisation for the architecture it is executed on. Furthermore, FOC's binary representation is compact, as we drop any alignment requirements; i.e., the code is bit-, not byte- or word-oriented. The structure of the code complicates the development of simple and efficient interpreters; this, however, is not regarded as a serious drawback as the approach aims at runtime compilation and not at interpretation.

The main contributions are the following: (1) a functional language that is attractive as an object code; (2) a structurally compressed binary code format;

and (3) an execution model for the functional object code. First experimental results concerning code compression are presented.

Section 2 discusses the view on architecture-independent code underlying this work. Section 3 introduces FOC's structure, type system, and its compressed binary format. Section 4 presents the execution model. Section 5 briefly reviews related work. Section 6 draws conclusions and outlines future work.

2 Architecture-Independent Code

Architecture independent code is no new idea, but the design space is large and people have used the basic concept for different purposes. This work takes the view that architecture-independent code essentially shifts work from what is traditionally considered to be compile time to runtime: Usually, carefully designed compilers use an intermediate language that interfaces the mainly architecture-independent analysis and optimisation parts of the compiler with a smaller architecture-dependent optimisation and code generation part. Now, consider splitting the compiler at exactly this interface and regarding the mentioned intermediate language as an architecture-independent object code. To obtain a complete system, we integrate the architecture-dependent optimisation and code generation part of the compiler into the runtime system.

Splitting an existing compiler in this way is usually infeasible, as there are subtle dependencies between the two parts, and because the intermediate language is not designed to be used as an object code; however, it should be clear that a compiler and an object code can be designed such that this redistribution of tasks is possible. Let us discuss in some detail the implications of the resulting compilation and execution model.

2.1 Portability and Mobility

Provided that the new object code is architecture-independent, the compiler gets perfectly portable.[1] This comes at the cost of a more complicated runtime system, but the implementation costs of the runtime system can easily be shared between different compilers, due to the clear interface provided by the new object code. However, the new work distribution may be considered more elegant, as it clearly separates architecture-dependent and -independent tasks.

However, more importantly the object code is also architecture-independent. Hence, the code is *mobile* in the sense that it can be transferred to and be run on any node of a heterogenous environment—multiple versions of binaries become superfluous. As communication is expensive, the size of the communicated object code is important. We will see that we can represent the architecture-independent

---

[1] This differs from approaches using C as an intermediate language or the recent work on `C--` [SP97]. These approaches try to modularize the compiler into an architecture-independent and an architecture-dependent part (a classical goal of compiler construction), but all this happens during what is classically regarded as compile time. In the case of FOC, classical compile-time work is actually delayed until runtime.

object code more space efficient than traditional binary code. We do not discuss the implications of mobility further, as the wide range of possible applications for mobile code were repeatedly explored in the context of the language Java.

2.2 Some Drawbacks

Unfortunately, mobile code not only leads to a more complex runtime system, but it also becomes more difficult to exploit existing techniques and tools. However, new tools, as for example VCODE [Eng96], that simplify the development of runtime systems including a code generator start to appear. Another point requires our attention: Moving architecture specific optimisations and code generation phases from the classical compilation to the runtime system increases program execution time. We have to minimize the additional time by an highly efficient implementation of the phases integrated into the runtime system; special properties of the mobile code help us to increase efficiency.

2.3 Smaller Code and More Accurate Optimisations

Two specific features of mobile code promise a reduction of the execution time to that of traditional architecture-dependent binaries, or even further down:

1. Mobile binaries are significantly smaller than architecture-dependent binaries. This saves I/O time during loading.
2. There is more information about the program at runtime than at compile time. This enables optimisations that are not possible during compile time.

Machine code, in particular RISC code, gets quite big. This is especially true for code generated for functional languages. Simply loading the binary of a program from disk or via the network from a local server takes time corresponding to a rather substantial amount of CPU cycles—and the ever increasing gap between I/O and processor time will make this effect even more pronounced in the future. Architecture-independent code can be much smaller and it can use the corresponding cycles for code generation [FK97]. It should be clear that this effect increases when we consider mobile code loaded from non-local networks, like the Internet. The mobile code FOC, proposed in this document, takes special measures to reduce code size below the size of more traditional byte codes, such as the code for the Java Abstract Machine (JAM) [LY96]. As the code is specific to the implementation of functional languages, further reduction is achieved.

Optimisation at runtime can be more effective than at compile time, as there is more information about the program. To start with, the exact specification of the executing architecture is known; in a parallel system, we know the number of processors; on a member of the x86 family, we exactly know which type of processor we have; and generally, we know the size of the caches, and so on. More importantly, regardless of separate compilation, at runtime, we have access to all interfaces and in principle to arbitrary inter-module information; this is especially significant for functional languages, where module boundaries hinder all sorts of analysis and optimisations (e.g., [HPS94,Boq95,BJ97]). Finally, there is no need to specialise the whole program in one go at load time; instead, initial specialisation as well as refining optimisations can be applied successively as need arises during the execution, and thus, code generation can exploit profiling information to concentrate on the few hot spots of the program.

2.4 Yet Another Mobile Code

The architecture-independent intermediate language Foc proposed in this paper significantly differs from existing proposals:

- It does not consist of a machine-language-like sequence of operations for a virtual stack or RISC machine; instead, it is a strongly-typed functional language encapsulating computations in monadic form.
- In contrast to byte-oriented codes, which are frequently used, Foc's binary representation is more compact as we drop any alignment requirements in the code, i.e., the code is bit-, not byte- or word-oriented.
- Functional languages require some implementation techniques, such as, tail-call optimisation and highly efficient garbage collection, that are not supported by other popular codes, like the JAM [LY96].

The structure of the code restricts its use to the compilation of languages with a functional origin; but therefore, the high-level representation allows significantly more radical optimisations than, for example, a pseudo-RISC code would—in particular, it allows more optimisations across module boundaries than are possible compile time. Furthermore, the implementation of optimisations as well as fast code generation is aided by the structure of the code, and moreover, strict safety checks on untrusted code are possible because of the comparatively simple control and data flow in combination with the full type information available in Foc. Finally, the high-level structure makes the code more compact, and Foc is meant to be specialised to the host architecture on execution, instead of being interpreted, which further differentiates it from many existing approaches.

3 The Definition of the Object Code

Next, we define the syntax of the new object code, discuss its semantics informally, and give some small examples. Afterwards, we have a look at the type system, and finally, at the binary representation. The present paper does not contain all details of Foc's language definition; these details can be found in [Cha98a].

3.1 Foc's Grammar

Foc is based on the second-order lambda calculus like GHC's intermediate language or XML [HM93]; thus, it is strongly typed. Type terms have the usual structure, where we write c `<`t_1`, ... , `t_n`>` for the application of a type constructor c to a sequence of types t_i. There are two special type constructors, denoted `<·>` and `[·]`, which represent computational and boxed types, respectively. A computation of type `<`τ`>` is a state transformer that yields a result of type τ—for state transformers, see, e.g., [PL95]. Computations are pointed, whereas all other types are unpointed—pointedness is discussed in [LP96]. The values of a boxed type `[`τ`]` are references to heap-allocated objects, whereas the values of other types are stored in-place—for unboxed values, see, e.g., [PL91]. Of particular interest are types of the form `[<`τ`>]`; they are boxed representations of computations—i.e., they are what is usually called a *thunk* or a *suspension* and they are needed to implement lazy evaluation [Pey92]. Finally, we allow universally quantified type variables, which are introduced by type terms of the form `\/`α`.`τ, introducing a variable α, visible in τ.

A module is a collection of datatype definitions and toplevel variable bindings. Datatype definitions are as usual; the following, for example, defines lists:

```
data List<a> = Nil () | Cons ([<[a]>], [<[List<a>]>]];
```

As mentioned, a type `[<`τ`>]` represents thunks; hence, `[<[a]>]` is a thunk that, when evaluated, produces `[a]`, i.e., a boxed value of type `a`. We omit a formal grammar for type terms and datatype definitions. Toplevel variable bindings associate lambda forms with variable names; they have the following grammar:

fdef → *id* `=` *lf* `;`
lf → `\` *um* *tvs* *vdefs* `:` *type* `->` *comp*
um → **upd**
| **nup**
tvs → `<` id_1 `,` ... `,` id_n `>`
vdefs → `(` id_1 `:` $type_1$ `,` ... `,` id_n `:` $type_n$ `)`

A lambda form [Pey92] represents a closure in a syntax resembling lambda abstraction. It begins with a backslash followed by an update mode *um*, which is `upd` if the closure has to be updated after evaluation (this is only relevant if the lambda form does not receive any arguments). Furthermore, a lambda form contains a list of local type variables *tvs* and a list of argument variables, which are typed. The type of the result of the lambda form follows after the colon; and finally, the body of the lambda form comes after the arrow; the body is used when the form is evaluated. The result of a function is required to be a computation, i.e., has to have a type of the form `<`τ`>`. Furthermore, when the update mode is `upd`, a boxed value has to be returned, i.e., the result type must have the form `<[`τ`]>`. As an example, consider the signature of the *map* function:

```
map = \nup <a,b> (f: [[<[a]>] -> <[b]>], l: [<[List<a>]>])
          : <[List<b>]> -> ⟨body of the lambda form⟩
```

The signature defines `map` to be polymorphic in `a` and `b`. According to the definition, it gets two arguments, `f` and `l`, where the first argument `f` is a boxed representation of a function (to be passed around, a function always has to be boxed). The function `f` expects to get a thunk that yields a boxed value of type `a`; `f` itself yields a computation resulting in a boxed value of type `b`. The second

```
comp  → return ( atom_1 , ... , atom_n )                 (continue with result)
      | ( id_1 , ... , id_n ) <- scomp_1 ; comp_2        (sequential composition)
      | id targs ( atom_1 , ... , atom_n )               (application)
      | case [box] id of { alts [dft -> comp ;] }        (selection)
scomp → comp_1 | ··· | comp_n                            (concurrent composition, n ≥ 1)
      | graph { id_1 = lf_1 ; ... id_n = lf_n ;}         (graph allocation, n ≥ 1)
targs → < type_1 , ... , type_n >
alts  → pat_1 -> comp_1 ; ... pat_n -> comp_n ;
pat   → lit                                              (literal pattern)
      | id targs [ id_1 : type_1, ... , id_n : type_n ]  (data closure pattern)
atom  → lit                                              (literal)
      | id                                               (variable)
```

Fig. 1. The syntax of computations.

argument, `l`, is a thunk resulting in a boxed list with elements of type `a`. Finally, map itself yields a computation producing a boxed list containing elements of type `b`. In other words, the variable `map` has type

```
[\/a.\/b. ([[<[a]>] -> <[b]>], [<[List<a>]>]) -> <[List<b>]>]
```

The syntax of function bodies is displayed in Figure 1. The nonterminal *id* refers to identifiers and *lit* to literals (such as, integers constants). The first two forms of computations represent the unit and bind operation of the computation monad; they return a result and perform sequential composition, respectively. The third and forth represent application and case selection, respectively. The latter has two variants: one for scrutinizing boxed and one for unboxed values.

On the right hand side of the arrow of a sequential composition, we have a simple computation, *scomp*, which is either a concurrent composition or graph allocation. Concurrent composition is a commutative monad operation used to express concurrency—similar in spirit to the *fork* combinator of [JH93]. Graph allocation introduces a set of mutually recursive associations of lambda forms with variables, like letrec expressions in functional programming; their operational interpretation is to allocate a heap closure for each lambda form and bind them to the local names—as letrec in the STGM [Pey92].

As an example of a complete function definition, consider `map` again; its definition is Haskell is the following:

$$\begin{array}{lcl} map & :: & (\alpha \to \beta) \to [\alpha] \to [\beta] \\ map\ f\ [] & = & [] \\ map\ f\ (x : xs) & = & f\ x\ :\ map\ f\ xs \end{array}$$

In Foc, it looks as follows:

```
map = \nup <a,b> (f: [[<[a]>] -> <[b]>], l: [<[List<a>]>])
          : <[List<b>]> ->
    (lv) <- l<> ();                   — demand the list head
    case box lv of {                  — scrutinize list head
```

```
Nil<a> [] -> Nil<b> ();
Cons<a> [x: [<[a]>], xs: [<[List<a>]>]] ->
  (fx) <- graph {fx = \upd <> () : <[b]> -> f<> (x);};
  (rmd) <- graph {          — thunk for recursive call
            rmd = \upd <> () : <[List<b>]> -> map<a,b> (f, xs);};
  Cons<b> (fx, rmd);};
```

After demanding the head of the list, it is scrutinized by the `case` expression. In the case of a `Cons` cell, two thunks are allocated for the head and tail of the resulting list, respectively. Note the separation of demand and case selection; this is a significant difference to the STG-language, which is motivated by the desire for fine-grained multi-threading and discussed in detail in [Cha98b,Cha97].

The following piece of code contains two concurrent invocations of the functions `f` and `g` that synchronize before their result is passed to a consumer `h`.

```
(x, y) <- f<> (a) | g<> (b);
h<> (x, y)
```

3.2 Type System

Figure 2 displays FOC's type system. It largely follows standard constructions, but some subtle points are worth noting. However, let us first review the notation briefly. In a type assertion $\Gamma \vdash e : \tau$, the expression e has type τ under the type environment Γ. The type environment initially contains the types of all primitives, data constructors, and toplevel variable bindings. The set of toplevel bindings is treated like the bindings in a `graph` construct. We use the notation $\overline{x}$ to abbreviate a sequence $x_1, \cdots, x_n$. We abuse this notation in two ways: First, $\backslash/\overline{\alpha}\,.\,\tau$ means $\backslash/\alpha_1\,.\;\ldots\;\backslash/\alpha_n\,.\,\tau$, and second, $\overline{v{:}\tau}$ means $v_1{:}\tau_1, \ldots, v_n{:}\tau_n$. Moreover, we apply type substitutions $\rho = [\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$ to a type τ by writing $\rho\tau$ and abbreviate them by $[\overline{\sigma/\alpha}]$. Furthermore, we regard multi-values $(\tau_1,\ \ldots\ ,\ \tau_n)$ as unlifted products and assume the following two equalities: $(\tau) = \tau$ and $(\tau_1,\ \ldots\ ,(\sigma_1,\ \ldots\ ,\sigma_m),\ \ldots\ ,\ \tau_n) = (\tau_1,\ \ldots\ ,\sigma_1,\ \ldots\ ,\sigma_m,\ \ldots\ ,\ \tau_n)$. The trivial inference rules for variables and literals are not included; the former does a lookup in the environment and the latter types literals with a primitive type.

In Rule (conc), n concurrent computations produce an n-tuple of results. Furthermore, a `graph` construct produces a tuple where each component corresponds to one binding, i.e., one allocated closure. These closures are boxed values; thus, the components τ_i of the produced tuple all have the form `[`$\cdots$`]`. Concurrent computation (conc) and graph allocation (graph) as well as the unit (ret) and bind operator (seq) of the computation monad <·> have a computational result type, which identifies them as operations in the computation monad.

Function application, in Rule (app), applies a function, i.e., a lambda form bound to a variable, a data constructor, or builtin operation, to a list of type arguments as well as a list of value arguments. Function application substitutes the universally quantified type variables by the types given as arguments, using the type substitution ρ. A single application supplies exactly one argument, which may be a multi-value. Passing arguments individually may sound inefficient, but

SIMPLE COMPUTATIONS

$$\frac{\Gamma \vdash c_1 : \texttt{<}\tau_1\texttt{>} \quad \cdots \quad \Gamma \vdash c_n : \texttt{<}\tau_n\texttt{>}}{\Gamma \vdash (c_1 \;\texttt{|}\; \cdots \;\texttt{|}\; c_n) : \texttt{<(}\tau_1\texttt{,}\ \ldots\ \texttt{,}\ \tau_n\texttt{)>}}\ \text{(conc)}$$

$$\frac{\begin{array}{ll} \Gamma' \cup \{\overline{v_1 : \sigma_1}\} \vdash e_1 : \sigma'_1 & \text{if } \pi_i = \mathtt{\textbf{upd}}, \sigma'_i = \texttt{<[}\xi\texttt{]>} \text{ for some type } \xi \\ \vdots & \text{where } \Gamma' = \Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \\ \Gamma' \cup \{\overline{v_n : \sigma_n}\} \vdash e_n : \sigma'_n & \qquad \tau_i = \texttt{[}\backslash\texttt{/}\overline{\alpha}_i\,\texttt{.}\,\texttt{(}\overline{\sigma}_i\texttt{)}\ \texttt{->}\ \sigma'_i\texttt{]}, 1 \leq i \leq n \end{array}}{\Gamma \vdash \mathtt{\textbf{graph}} \left\{ \begin{array}{c} x_1 = \backslash\pi_1\ \texttt{<}\overline{\alpha}_1\texttt{>}\ \texttt{(}\overline{v_1 : \sigma_1}\texttt{)}\ :\ \sigma'_1\ \texttt{->}\ e_1\texttt{;} \\ \vdots \\ x_n = \backslash\pi_n\ \texttt{<}\overline{\alpha}_n\texttt{>}\ \texttt{(}\overline{v_n : \sigma_n}\texttt{)}\ :\ \sigma'_n\ \texttt{->}\ e_n\texttt{;} \end{array} \right\} : \texttt{<(}\tau_1\texttt{,}\ \ldots\ \texttt{,}\ \tau_n\texttt{)>}}\ \text{(graph)}$$

COMPUTATIONS

$$\frac{\Gamma \vdash a_1 : \tau_1 \quad \cdots \quad \Gamma \vdash a_n : \tau_n}{\Gamma \vdash \mathtt{\textbf{return}}\ \texttt{(}a_1\texttt{,}\ \ldots\ \texttt{,}\ a_n\texttt{)} : \texttt{<(}\tau_1\texttt{,}\ \ldots\ \texttt{,}\ \tau_n\texttt{)>}}\ \text{(ret)}$$

$$\frac{\Gamma \vdash s : \texttt{<(}\tau_1\texttt{,}\ \ldots\ \texttt{,}\ \tau_n\texttt{)>} \qquad \Gamma \cup \{v_1 : \tau_1, \ldots, v_n : \tau_n\} \vdash c : \texttt{<}\tau\texttt{>}}{\Gamma \vdash \texttt{((}v_1\texttt{,}\ \ldots\ \texttt{,}\ v_n\texttt{)}\ \texttt{<-}\ s\texttt{;}\ c\texttt{)} : \texttt{<}\tau\texttt{>}}\ \text{(seq)}$$

$$\frac{\Gamma \vdash f : \texttt{[}\backslash\texttt{/}\overline{\alpha}\,\texttt{.}\,\texttt{(}\tau_1\texttt{,}\ \ldots\ \texttt{,}\ \tau_n\texttt{)}\ \texttt{->}\ \tau\texttt{]} \qquad \Gamma \vdash a_1 : \rho\tau_1 \quad \cdots \quad \Gamma \vdash a_n : \rho\tau_n \text{ with } \rho = [\overline{\sigma}/\overline{\alpha}]}{\Gamma \vdash f\ \texttt{<}\overline{\sigma}\texttt{>}\ \texttt{(}a_1\texttt{,}\ \ldots\ \texttt{,}\ a_n\texttt{)} : \rho\tau}\ \text{(app)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathtt{\textbf{case}}\ v\ \mathtt{\textbf{of}}\ \texttt{\{}\mathit{alts}\texttt{\}} : \tau \\ \Gamma \vdash v : \sigma \quad \Gamma \vdash l : \sigma \quad \Gamma \vdash c : \tau \end{array}}{\Gamma \vdash \mathtt{\textbf{case}}\ v\ \mathtt{\textbf{of}}\ \texttt{\{}l\ \texttt{->}\ c\texttt{;}\ \mathit{alts}\texttt{\}} : \tau}\ \text{(case-lit)} \qquad \frac{\Gamma \vdash c : \tau}{\Gamma \vdash \mathtt{\textbf{case}}\ [\mathtt{\textbf{box}}]\ v\ \mathtt{\textbf{of}}\ \texttt{\{}\mathtt{\textbf{dft}}\ \texttt{->}\ c\texttt{;}\ \mathit{alts}\texttt{\}} : \tau}\ \text{(case-dft)}$$

$$\frac{\begin{array}{ll} \Gamma \vdash d : \texttt{[}\backslash\texttt{/}\overline{\alpha}\,\texttt{.}\,\texttt{(}\overline{\sigma'}\texttt{)}\ \texttt{->}\ \sigma\texttt{]} & \Gamma \cup \{\overline{v : \rho\sigma'}\} \vdash c : \tau \text{ with } \rho = [\overline{\sigma''}/\overline{\alpha}] \\ \Gamma \vdash \mathtt{\textbf{case}}\ [\mathtt{\textbf{box}}]\ v\ \mathtt{\textbf{of}}\ \texttt{\{}\mathit{alts}\texttt{\}} : \tau & \Gamma \vdash v : \rho\sigma, \text{ or } \Gamma \vdash v : \texttt{[}\rho\sigma\texttt{]} \text{ if } \mathtt{\textbf{box}} \text{ present} \end{array}}{\Gamma \vdash \mathtt{\textbf{case}}\ [\mathtt{\textbf{box}}]\ v\ \mathtt{\textbf{of}}\ \texttt{\{}d\ \texttt{<}\overline{\sigma''}\texttt{>}\ \texttt{[}\overline{v : \rho\sigma'}\texttt{]->}\ c\texttt{;}\ \mathit{alts}\texttt{\}} : \tau}\ \text{(case-con)}$$

Fig. 2. Typing rules

the compiler is expected to use multi-values when a function is called often with more than one argument—wrapper functions are used for curried applications. For example, the following definition of `map1` is a wrapper used when a curried version of the previously discussed `map` is needed:

```
map1 = \nup <a,b> (f: [[<[a]>] -> <[b]>])
             : [[<[List<a>]>] -> <[List<b>]>] ->
    (papp) <- graph {
                 papp = \nup <> (l: [<[List<a>]>]) : <[List<b>]> ->
```

```
                    map<a,b> (f, l);};
    return (papp);
```

It allocates a partial application, called `papp`, containing the first argument to `map`, namely `f`. The wrapper may be inlined, to save the call to `map1`. On the one hand, this scheme is less efficient than the push-enter model used in the STGM [Pey92]—when a function is applied to a number of arguments individually in a tail call sequence. On the other hand, it avoids problems associated with a stack of pending arguments, including a simplification of thread scheduling (see [Cha98b] for a related execution model, which sticks to STGM's push-enter model).

The Rules (case-lit), (case-dft), and (case-con) handle the three kinds of alternatives occurring in `case` expressions. A single case expression can only contain alternatives matching literals or otherwise alternatives matching data constructors. Furthermore, the latter require to instantiate the universally quantified types of the data constructor, similar to Rule (app).

3.3 Compressed Binary Code

A FOC object code file is essentially a compact representation of the structure tree of a module. The design of the binary format has two objectives: The code should be compact and it should be easy to read for the runtime system. Both objectives ultimately minimize the time needed to load the object code, and thus, maximize the time available for code generation at runtime. The functional origin of the code already makes it smaller than architecture-dependent machine code. Moreover, FOC uses three techniques to reduce the size further: (1) unaligned code, (2) minimized symbol information, and (3) a type compression scheme.

The code is unaligned as it does not have a fixed division into bytes or words; instead, each data item is represented by the minimal number of bits without padding. For example, to represent a value in the range from 0 to 3, two bits are sufficient in FOC, where a byte-aligned code would waste six bits for padding.

The second technique minimizes the space used for identifiers. First of all, lexemes of identifiers, apart from those exported, are omitted in the code. Instead, at the defining position, each identifier is implicitly associated with a number, called the *identifier code*; at each usage position of the identifier, its code is used to reference the definition. Moreover, we exploit lexical scoping to minimize the range of values used for identifier codes. In combination with the use of unaligned code, this minimizes the number of bits required to encode identifiers. For example, in a position where less than 32 variable identifiers are visible, we need only 5 bit to encode an identifier. The structure of FOC allows us to use separate name spaces for type names, top level function definitions, and local variables; hence, the often high number of imported functions does not diminish the advantages of the compressed format. The use of identifier codes, moreover, has the advantage of eliminating the need for sophisticated symbol management and any kind of name analysis or renaming procedure in the runtime system.

FOC binaries are fully typed, which is necessary for some optimisations (like type specialisation) and allows tighter safety checks on untrusted modules. Without any provisions, full type information would increase the size of the code considerably. We reduce this effect by a technique related to common subexpression elimination. Whenever a type term occurs for the first time, it is implicitly associated with a *type code*. Whenever the term occurs again, the type code replaces the term itself. This method is applied not only to complete terms, but also to all subterms. FOC's binary code format is defined in detail in [Cha98a].

First experiments with a code generator for FOC support the effectiveness of the described method. For example, a Haskell program of about 750 byte (no comments included) that does some tree traversals and list processing results in less than 300 byte architecture-independent FOC code, whereas the object file generated by the Glasgow Haskell Compiler (GHC), version 3.02, for a Pentium processor is about 3500 byte (the object file is in Linux ELF format; symbols were removed with `strip`). GHC's code compressed with `gzip` still is over 1200 byte and even the compressed Haskell source has over 300 byte.

4 Execution Model

The execution model has its roots in the STGM [Pey92]. It deviates from the STGM in its integration of multi-threading (as introduced in [Cha97]) as well as the omission of a stack for pending arguments (as mentioned earlier). The operational semantics is defined by a transition system, which is presented here only in a simplified form; in particular, we omit the treatment of multiple instances modeling distribution and communication. The integration of distribution and communication, a discussion of the mapping of this model to concrete distributed machines, many more details, and the interpretation of the transition rules as syntactic sugar for an interpreter in Haskell are covered in [Cha97].

The state of the transition system, consists of a thread pool Γ, a control stack ds, a waiting list ws, and a heap h. Both the control stack and the waiting list contain thread identifiers. We use the control stack to schedule threads, as far as this is possible, in a depth-first order; the threads in the waiting list were suspended. A thread has the form $\langle \tau \bullet \gamma, ts \rangle$, where τ is the thread identifier, γ the executed code form, and ts, called the target list, specifies the destination of the result computed by the thread. There are six different code forms: *Exec* c ρ executes the computation c under the local environment ρ; *Enter* o ws enters the closure named o with the arguments ws; *Ret* ws returns the values ws to locations defined in the thread's target list; *Sync* n ρ γ marks a suspended thread that waits for n values that have to be put into its local environment ρ before it continues with the code form γ; *Suspend* t g ρ suspends the current thread, which will enter the closure named g when thread t resumes it; and *Poll* gets all runnable threads from the waiting list and puts them on the control stack. There are two kinds of targets, which are used by *Ret* to determine the locations where the returned value is expected: *Upd* o updates the closure named o

with the result and $\mathit{Store}\ \tau\ \mathit{xs}$ associates the result with the variables xs in the environment of the thread named τ, which is required to have a *Sync* code form.

The heap associates names with closures; a closure is a lambda form without any type information, a code form instead of a the body computation, and add an environment of its free variables. A closure is written $(\mathit{xs} \setminus\pi\ \mathit{vs}\ \rightarrow\ \gamma)\ \mathit{ws}$, where xs are the closure's free variables, π the update mode, vs its function arguments, γ the code form, and ws the environment—similar to [Pey92] and [Cha97].

In the rules, we omit types. The notation $\rho \uparrow x$ denotes the lookup of x in the environment ρ, and $\nu\ \rho\ \sigma\ x$ looks x up in ρ first and, if not found, looks it up in the global environment σ, which contains all toplevel definitions. The function ν and the notation $[x \mapsto w]$, associating x with w, are also lifted to sequences.

Returning a result. A return is handled by the code form *Ret*.

$$\langle \delta \bullet \mathit{Exec}\ (\texttt{return}\ \mathit{as})\ \rho,\ \mathit{ts}\rangle \in \Gamma \quad \cdots$$

$$\overset{\sigma}{\Longrightarrow} \langle \delta \bullet \mathit{Ret}\ (\nu\ \rho\ \sigma\ \mathit{as}) \qquad ,\ \mathit{ts}\rangle \in \Gamma \quad \cdots$$

Each $\mathit{Upd}\ o$ target overwrites the closure o with an indirection to the returned closure (only possible for boxed results, which are always unary). Furthermore, each $\mathit{Store}\ \tau\ \mathit{xs}$ target decrements the synchronization counter of the *Sync* form contained in thread τ and associates the returned values with the variables xs in the *Sync* form's environment ρ.

$$\langle \delta \bullet \mathit{Ret}\ (o),\ (\mathit{Upd}\ o') : \mathit{ts}\rangle \in \Gamma \quad \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

$$\overset{\sigma}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Ret}\ (o), \qquad\qquad \mathit{ts}\rangle \in \Gamma \quad \delta : \mathit{ds} \quad \mathit{ws} \quad h'$$

where

$$h' = [o' \mapsto\ (\{\underline{x}\}\ \texttt{\textbackslash nup}\ ()\ \rightarrow\ \mathit{Exec}\,(x\ ()))\ \{o\}] \rhd h$$

$$\langle \delta \bullet \mathit{Ret}\ \mathit{ws},\ (\mathit{Store}\ \tau\ \mathit{xs}) : \mathit{ts}\rangle \in \Gamma \quad \cdots$$

when $\langle \tau \bullet \mathit{Sync}\ n\ \rho\ \gamma, \mathit{ts}_s\rangle \in \Gamma$

$$\overset{\sigma}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Ret}\ \mathit{ws}, \qquad\qquad \mathit{ts}\rangle \in \Gamma' \quad \cdots$$

where

$$\Gamma' = [\langle \tau \bullet \mathit{Sync}\,(n-1)\,([\mathit{xs} \mapsto\ \mathit{ws}] \rhd \rho)\ \gamma, \mathit{ts}_s\rangle] \rhd \Gamma$$

$$\langle \delta \bullet \mathit{Ret}\ \mathit{ws},\ []\rangle \in \Gamma \quad \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

$$\overset{\sigma}{\Longrightarrow} \qquad\qquad \Gamma \qquad \mathit{ds} \quad \mathit{ws} \quad h$$

Application. We omit the application of primitive functions. The application of a data constructor allocates a constructor closure and returns a reference o to the new closures. Underlines names are fresh; vs is a sequence of fresh names.

$$\langle \delta \bullet \mathit{Exec}\ (d\ \ \mathit{as})\ \rho,\ \mathit{ts}\rangle \in \Gamma \quad \delta : \mathit{ds} \quad \mathit{ws} \quad h$$

when d is a data constructor

$$\overset{\sigma}{\Longrightarrow} \quad \langle \delta \bullet \mathit{Ret}\ (o) \qquad ,\ \mathit{ts}\rangle \in \Gamma \quad \delta : \mathit{ds} \quad \mathit{ws} \quad h'$$

where

$$h' = [\underline{o}\ \mapsto\ (\underline{\mathit{vs}}\ \texttt{\textbackslash nup}\ ()\ \rightarrow\ \mathit{Exec}\,(d\ \mathit{vs}))\ (\nu\ \rho\ \sigma\ \mathit{as})] \rhd h$$

Application of a variable f enters the referenced closure $\nu\ \rho\ \sigma\ f$. A closure's body γ is a functional that given the environment $([xs \mapsto\ us]\ +\!\!+\ [vs \mapsto\ ws])$, which binds the arguments and the closure's environment, becomes a code form.

$$\begin{array}{lllll}
 & \langle \delta \bullet \mathit{Exec}\ (f\ \ as)\ \rho \qquad\qquad ,\ ts\rangle \in \Gamma & \cdots & & \\
\text{when } f \text{ is a variable name} & & & & \\
\overset{\sigma}{\Longrightarrow} & \langle \delta \bullet \mathit{Enter}\ (\nu\,\rho\,\sigma\,f)\ (\nu\,\rho\,\sigma\ as),\ ts\rangle \in \Gamma & \cdots & &
\end{array}$$

$$\begin{array}{lllll}
 & \langle \delta \bullet \mathit{Enter}\ o\ ws \qquad\qquad\qquad ,\ \ ts\rangle \in \Gamma & \delta : ds & ws & h \\
\text{when } h \uparrow o = \ (xs \setminus\!\pi\ vs\ \ \to\ \gamma)\ us & & & & \\
\overset{\sigma}{\Longrightarrow} & \langle \delta \bullet \gamma\ ([xs \mapsto us] +\!\!+ [vs \mapsto ws]),\ ts'\rangle \in \Gamma & \delta : ds & ws & h' \\
\text{where} & & & & \\
\quad h' \ = \ [o\ \mapsto\ (\{\underline{t}\}\ \backslash\mathbf{nup}\ ()\ \to\ \mathit{Suspend}\,t\ o)\ \{\delta\}]\ \triangleright\ h & & & & \\
\quad ts' \ = \ \text{if } \pi = \mathbf{nup} \text{ then } ts \text{ else } (\mathit{Upd}\ o) : ts & & & &
\end{array}$$

In case of an update closure, an *Upd* is added to the target list ts'. Furthermore, the entered closure o is overwritten with a $\mathit{Suspend}\ \rho$, which is, strictly speaking, only necessary in the case of update closures—but, we do not make this optimisation here, to keep the rule simple.

Selection. We omit selection as it is does not give new insights.

Graph allocation. A `graph` construct allocates a closure for each of its lambda forms. The environment of the closure is computed from the free variables in the body of the lambda form, Fv c_i, and the bindings of these variables in the local environment ρ plus the bound variables vs' (the bindings are mutually recursive).

$$\begin{array}{lllll}
 & \langle \delta \bullet \mathit{Exec}\ (vs\ \texttt{<- graph \{}\ldots\texttt{;}\,v_i'\ \texttt{=}\ \backslash\pi\ \ as_i\ \texttt{->}\ c_i\,\texttt{;}\ \ldots\texttt{\};}\ \ c)\ \rho,\ ts\rangle \in \Gamma & \delta : ds & ws & h \\
\overset{\sigma}{\Longrightarrow} & \langle \delta \bullet \mathit{Exec}\ c\ (\rho +\!\!+ [os/vs]) \qquad\qquad\qquad\qquad ,\ ts\rangle \in \Gamma & \delta : ds & ws & h' \\
\text{where} & & & & \\
\quad h' \ = \ [\ldots,\ \underline{o_i}\ \mapsto\ (\mathrm{Fv}\ c_i \setminus\!\pi\ as_i\ \to\ \mathit{Exec}\,c_i)\ ((\rho +\!\!+ [os/vs'])\ (\mathrm{Fv}\ c_i))),\ \ldots]\ \triangleright\ h & & & &
\end{array}$$

Here o_i is an example from os and v_i' an example from vs'. Note that the code form *Exec* is only partially applied—the environment is missing. This environment is supplied by the transition rules handling *Enter*.

Concurrency. Concurrent composition introduces a thread δ' for each of the concurrently composed computations c_i. The computation c after the semicolon must wait until each of these threads produced a result. This is achieved by the use of $\mathit{Sync}\ n\ \rho\ (\mathit{Exec}\ c)$, which waits for the execution of the n targets of the form $\mathit{Store}\ \delta\ vs_i$ contained in the target list of each of the new threads. Each of these *Store* targets potentially returns a multi-value.

$$\begin{array}{lllll}
 & \langle \delta \bullet \mathit{Exec}\ ((vs_1, \ldots, vs_n)\ \texttt{<-}\ c_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n; c)\ \rho,\ ts\rangle \in \Gamma & \delta : ds & ws & h \\
\overset{\sigma}{\Longrightarrow} & \langle \delta \bullet \mathit{Sync}\ n\ \rho\ (\mathit{Exec}\ c) \qquad\qquad\qquad ,\ ts\rangle \in \Gamma' & ds' +\!\!+ (\delta : ds) & ws & h \\
\text{where} & & & & \\
\quad \Gamma' \ = \ [\langle \delta' \bullet \mathit{Exec}\,c'\ \rho, [\mathit{Store}\,\delta\ vs_i]\rangle & & & & \\
\qquad\qquad \mid\ (vs',\ c',\ \delta')\ \leftarrow\ \mathit{zip3}\,[vs_1,\ \ldots,\ vs_n]\,[c_1,\ \ldots,\ c_n]\ \underline{ds'}]\ \triangleright \Gamma & & & &
\end{array}$$

The identifiers ds' of the new threads are placed on top of the control stack, i.e., they are executed before an attempt is made to execute δ again. If all *Store* targets were executed, δ's *Sync* target will have a 0 in its first arguments—and the immediately following rule applies, which executes the body γ. However, if any of the threads got suspended, some results may be outstanding, in which case $n \geq 1$ and δ suspends, i.e., is put into the waiting list.

$$\begin{array}{lll}
 & \langle \delta \bullet \mathit{Sync}\ 0\ \rho\ \gamma,\ \mathit{ts} \rangle \in \Gamma & \cdots \\
\stackrel{\sigma}{\Longrightarrow} & \langle \delta \bullet \gamma\ \rho \qquad\ ,\ \mathit{ts} \rangle \in \Gamma & \cdots
\end{array}$$

$$\begin{array}{lllll}
 & \langle\ \delta \bullet \mathit{Sync}\ n\ \rho\ \gamma,\ \mathit{ts} \rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\
\text{when } n \geq 1 & & & & \\
\stackrel{\sigma}{\Longrightarrow} & \langle \underline{\delta'} \bullet \mathit{Poll} \qquad ,\ [] \rangle \in \Gamma & \delta' : \mathit{ds} & \delta : \mathit{wd} & h
\end{array}$$

In place of the suspended *Sync* thread, a *Poll* thread is created. *Poll* fetches all runnable threads out of the waiting list and puts them on top of the control stack. A thread becomes runnable when its *Sync* counter eventually reaches zero, as outstanding targets are executed.

$$\begin{array}{lllll}
 & \langle \delta \bullet \mathit{Poll},\ [] \rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\
\stackrel{\sigma}{\Longrightarrow} & \Gamma & (\mathit{filter}\ \mathit{runnable}\ \mathit{ws}) +\!\!+\ \mathit{ds} & \mathit{filter}\ (\mathit{not} \circ \mathit{runnable})\ \mathit{ws} & h
\end{array}$$

Suspension. The last rule handles the *Suspend* form introduced when entering a closure. A thread executing *Suspend* is placed in the waiting list. It acquires a *Sync* form waiting for a single *Store* δ (). The second argument () indicates that the store does not really store any result, but merely signals that the thread τ, which evaluates the blocking closure, finished. When this happens, δ re-enters the closure, which was meanwhile updated by another target of τ.

$$\begin{array}{lllll}
 & \langle \delta \bullet \mathit{Suspend}\ t\ g\ \rho \qquad ,\ \mathit{ts} \rangle \in \Gamma & \delta : \mathit{ds} & \mathit{ws} & h \\
\multicolumn{5}{l}{\text{when } \rho \uparrow t = \tau \text{ with } \langle \tau \bullet \gamma, \mathit{ts}_t \rangle} \\
\stackrel{\sigma}{\Longrightarrow} & \langle \delta \bullet \mathit{Sync}\ 1\ []\ \mathit{Enter}\ g\ (),\ \mathit{ts} \rangle \in \Gamma' & \mathit{ds} & \delta : \mathit{ws} & h \\
\multicolumn{5}{l}{\text{where}} \\
\multicolumn{5}{l}{\quad \Gamma' = [\langle \tau \bullet \gamma, \mathit{Store}\ \delta\ () : \mathit{ts}_t \rangle] \triangleright \Gamma}
\end{array}$$

To be precise, we need a rule that creates a *Poll* thread when the control stack *ds* gets empty. It is omitted here together with the treatment of multiple distributed and parallel machine instances, each with its own thread pool, control stack, waiting list, and heap, as well as asymmetric communication by thread posting. These features can be realized as described in [Cha97].

5 Related Work

Foc, like many other mobile codes [FK97,LY96], is language dependent. This does not necessarily mean that the code can only be used for a single language, but it is restricted to a language family—in our case, functional languages. However, there are also attempts to define language-independent mobile codes, such as the object code of the Inferno operating system [BL98] and OmniVM [ATLLW96]. Such codes are attractive as the work invested in the runtime code-generator can be shared between many compilers. Nevertheless, such generality usually implies that the codes is more low-level, has less structure, and less accurate type information. As a result, the code is less compact, safety checks are more difficult, and some optimisations become more difficult or even impossible.

David Wakeling has done some work on achieving architecture-independence by compiling Haskell to the JAM [Wak98]. Unfortunately, it seems rather difficult to achieve good performance for a functional language with the JAM; this was also found in first attempts to compile ML to the JAM. This is mainly because the JAM is designed for Java and does not handle memory allocation and tail recursion efficiently enough for a functional language.

Runtime code-generation for a functional language was also used by Wakeling [Wak95] to reduce the memory footprint by throw-away compilation, i.e., runtime-generated code that is cached, but partially discarded when memory becomes scarce. Foc shares with Wakeling's approach that in both cases the intermediate language of a functional language compiler is used as an object code—in Wakeling's case, a variant of the G-machine code. However, Wakeling's objectives are somehow different from those in Foc's design, as the latter does not care much about the memory footprint, but aims at achieving code mobility with minimal compromises in efficiency. Nevertheless, it might be worthwhile to investigate whether Foc can be also be used for Wakeling's purpose.

The idea of compensating for overhead from runtime code generation by reducing the size of the object code was inspired by Franz and Kistler [FK97]. However, their work is concerned with object-oriented languages and, in particular, Oberon. Their object code also represents the structure tree of an intermediate language, but they apply a dictionary compression method to the whole module. There is other work [EEF$^+$97] on code compressions that even tries to directly execute the compressed code. Foc's object code format is much simpler and significant size reduction is achieved by choosing a comparatively high-level, language-dependent object code.

From the perspective of an intermediate language for compiling a functional language, Foc is in some sense in between the Stg-language [Pey92] and GRIN [BJ97]. Both GRIN and Foc are monadic and try to capture more structure than a stack-machine code (e.g., the G-machine code [Pey87]), while being more explicit than previous structured codes, like the STG-language [Pey92]. But in contrast to GRIN, Foc is higher-order, typed, supports concurrency and distribution, and takes a different approach to forcing thunks etc. The structure of Foc was partly inspired by the search for an intermediate language that provides some of the benefits of CPS [App92] without its deficiencies [HD94,FSFD93]. The goal was to be as low-level as possible, while still remaining functional.

The treatment of boxed versus unboxed types in Foc was largely taken from [Cha97, Chapter 3], but without considering kinds. It would be interesting to see, how close Foc is to the goal of [PLST98], i.e., whether it is a viable basis for both lazy and strict languages.

6 Conclusion

It was the purpose of this paper to outline an approach towards a mobile, yet efficient implementation of functional languages. Whether the overall efficiency will be sufficient remains to be seen, but first numbers show, in comparison to architecture-dependent code, a reduction in code size of an order of magnitude. Currently, a runtime system for Foc is build with the intention to reuse GHC's runtime system for services such as storage management, profiling, and so on. The runtime code generator leaves room for plenty of research. Optimisations are only worthwhile if they pay off including the time needed for optimising; however, the presence of all modules of a program and profiling information simplifies some optimisations and makes some more powerful.

Acknowledgements. I am very grateful for the interesting and inspiring discussions with Simon L. Peyton Jones at IFL'98; they led to significant simplification and overall improvement of Foc. I am also happy to thank Gabriele Keller for her helpful comments on the paper.

References

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[ATLLW96] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 127–136. ACM Press, 1996.

[BJ97] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Selected papers from the 8th International Workshop on Implementation of Functional Languages*, number 1268 in Lecture Notes in Computer Science, Berlin, Germany, 1997. Springer-Verlag.

[BL98] Lucent Technologies Bell Labs. Inferno. http://www.lucent-inferno.com/, 1998.

[Boq95] Urban Boquist. Interprocedural register allocation for lazy functional languages. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 1995.

[Cha97] Manuel M. T. Chakravarty. *On the Massively Parallel Execution of Declarative Programs.* PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1997.

[Cha98a] Manuel M. T. Chakravarty. Foc: A mobile functional object code. Internal Report (http://www.score.is.tsukuba.ac.jp/~chak/haskell/), 1998.

[Cha98b] Manuel M. T. Chakravarty. Lazy thread and task creation in parallel graph-reduction. In Chris Clark, Kevin Hammond, and Tony Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL '97, Selected Papers*, number 1467 in Lecture Notes in Computer Science, pages 231–249, Berlin, 1998. Springer Verlag.

[EEF$^+$97] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *SIGPLAN '97 Conference on Programming Language Design and Implementation.* ACM Press, 1997.

[Eng96] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, pages 160–170. ACM Press, 1996.

[FK97] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, 1997.

[FSFD93] Cormac Flanagan, Amr Sabry, Matthias Felleisen, and Bruce Duba. The essence of compiling with continuations. In *PLDI '93*. ACM Press, 1993.

[HD94] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1994.

[HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[HPS94] Cordelia Hall, Simon L. Peyton Jones, and Patrick M. Sansom. Unboxing using specialisation. In *Glasgow Functional Programming Workshop*, Workshops in Computing. Springer-Verlag London, 1994.

[JH93] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Research Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, 1993.

[LP96] John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *Proc European Symposium on Programming*, 1996.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall, 1987.

[Pey92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), 1992.

[PL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the International Conference on Functional Programming and Computer Architecture*, 1991.

[PL95] Simon Peyton Jones and John Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[PLST98] S. L. Peyton Jones, J. Launchbury, M. B. Shields, and A. P. Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *Proceedings of POPL'98*, 1998.

[SP97] Dino Oliva Simon Peyton Jones, Thomas Nordin. C--: a portable assembly language. In *International Conference on Functional Programming*, 1997.

[Wak95] David Wakeling. A throw-away compiler for a lazy functional language. In *Mt Fuji International Workshop on Functional and Logic Programming*, Singapore, 1995. Japan Society of Software Science and Technology, World Scientfic Publishing.

[Wak98] David 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)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.