IR-MetaOCaml: (re)implementing MetaOCaml

by

Evgeny Roubinchtein

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2015

Abstract

Multi-stage programming is a form of metaprogramming that is an extension of ideas and techniques of partial evaluation. This thesis discusses a (re)implementation of a multi-stage programming system MetaOCaml. The system presented here differs from the OCaml implementation by Taha et al in that it is implemented on top of a modern OCaml compiler. It differs from BER MetaOCaml in that it supports generation of native code in a turn-key fashion. It differs from both systems in that it uses the OCaml intermediate representation to represent the notion of code (to the best of my knowledge, existing system use abstract syntax trees instead.)

Preface

The thesis is original, unpublished, independent work by the author, Evgeny Roubinchtein.

Table of Contents

List of Tables

List of Figures

Glossary

AST abstract syntax tree , an (in general n-ary) tree representing the syntactical structure of the program, (usually) produced by a compiler's parser

Chapter 1

Introduction

When faced with a problem that can be solved by computer, one possible approach is to write a program that solves that problem. Another approach is to write a program that produces a program that solves the problem. The former approach is commonly known as programming The latter is (somewhat less commonly) known as meta-programming. Practitioners of meta-programming report being able to produce concise and expressive programs [11] [14]; being able to produce programs that run efficiently with relatively little programmer effort has also been reported [16] [20][2].[1]

Multi-stage programming is a form of meta-programming which builds on ideas from partial evaluation. Figure 1.1 shows the flow of data in a typical partial evaluation system: a program together with a specification of some of its inputs is first subjected to (automated) *binding time analysis* to decide which of the program's internal variables depend on the given inputs. The binding time analysis phase is followed by partial evaluation proper, which produces a *residual program*. When the residual program is run, any values whatsoever may be assigned to the subset of the inputs that were not specified during partial evaluation. If S denotes the subset of the original program's inputs that were specified during partial evaluation and U denotes the subset of program's inputs that were not specified during

[1] While hand-coding the program in assembly and optimizing it for a specific architecture might produce a significantly more efficient program, part of the emphasis here is on saving programmer's effort.

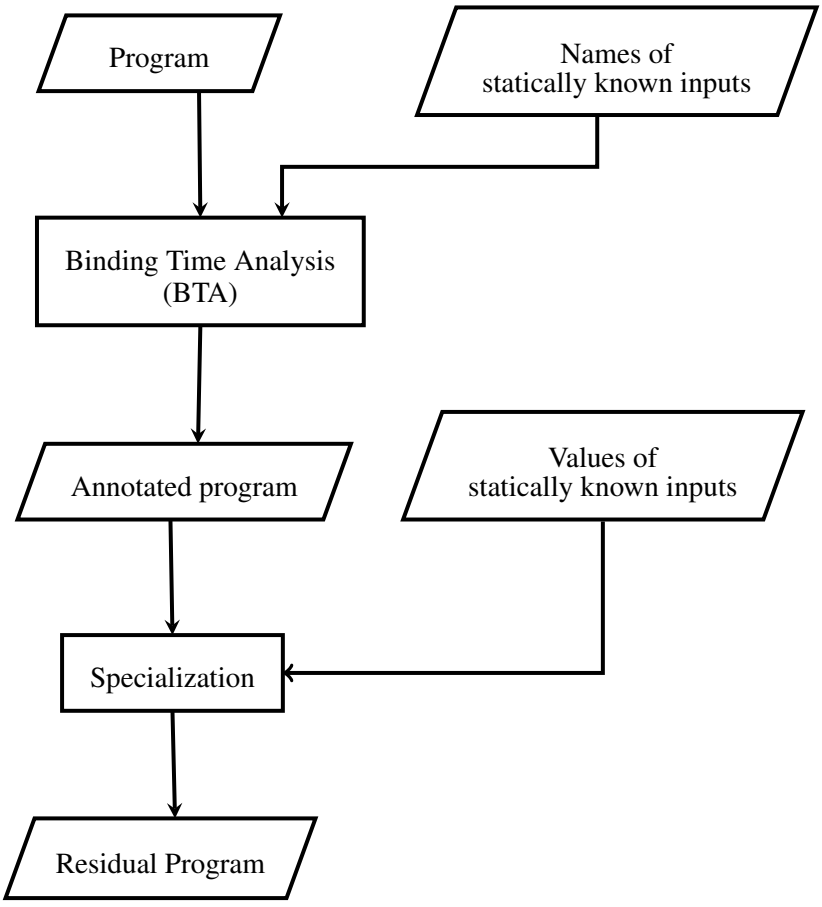

Figure 1.1: Partial evaluation data flow: the language of annotated and residual program need not be the same.

partial evaluation, then running the residual program while assigning any values whatsoever to the inputs in U is equivalent to running the original program with the values that were given to the partial evaluator for the inputs in S and providing those same – arbitrary – values for the inputs in U.

Figure 1.2 shows the flow of data in a multi-stage programming system: the responsibility for doing binding time analysis is transferred from the machine to the human: programmer is responsible for *annotating* the program to describe which values may be immediately calculated from a given set of inputs. The machine then faithfully follows the annotations provided by the programmer to arrive at a

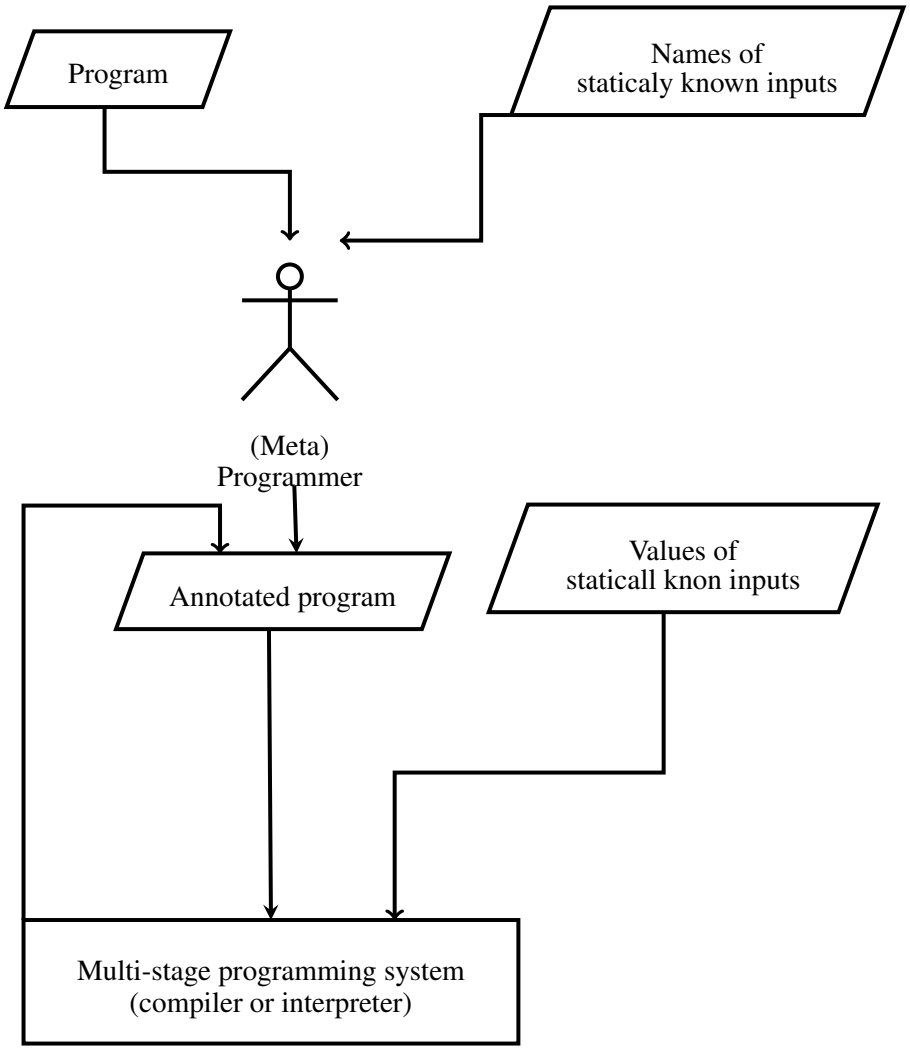

Figure 1.2: Multi-stage programming data flow: the language of annotated and residual program is the same, and specialization step may be repeated an arbitrary number of times.

residual program.

Lisp macro system may be considered as an early example of a multi-stage programming system. Graham [14] popularized meta-programming with Lisp macros, and also observed that, "macro definitions are harder to read than ordinary code" [15]. A plausible approach to help programmers debug Lisp macros would be to have

problematic usages of macros. *Types* are a programming languages device that enables compile-type checking of programs. If it were possible to create a type system for Lisp macros, then compile-type checking of Lisp macros could be implemented. Wand studied the Lisp macro system to determine if any guarantees could be made about the behavior of Lisp macros, and concluded that no type system could be devised for the Lisp macros system.[33]

MetaML, invented by Sheard [31], is a typed meta-programming system. The type system detects certain programming errors. Examples of the kinds of errors prohibited by the type system include unbound variables, and using a variable *in the wrong stage* (before its value has been calculated). Type systems for multi-stage programming detect those errors without running the program, and without needing to re-examine the program during each successive stage of the evaluation. Taha showed that, unlike Lisp macro system, a type system for MetaML can be devised [29].

MetaOCaml is a descendant of MetaML, first implemented by Taha [4]. Taha's work was done on OCaml version 3, and was never (to my knowledge) ported to OCaml version 4. BER MetaOCaml is an implementation of MetaOCaml for OCaml version 4 that supports generating byte code (only) in a turn-key fashion. The work presented here is a clean-room re-implementation of MetaOCaml with a focus on native code generation.

The goal of this project was to explore using the OCaml intermediate representation[2] rather than AST as the concrete representation of the *program* that is being manipulated by the meta-programming system. To the best of my knowledge, all existing MetaOCaml implementations manipulate ASTs. Because of how the OCaml compiler is implemented, manipulating ASTs means that program is type-checked at every stage of evaluation. While not harmful, such type checking is unnecessary in view of the existence of multi-stage programming type systems. We were curious if getting rid of the repeated type checking would positively impact run-time performance. The performance evaluation of the resulting system is found in Chapter 4.

A secondary goal of the project is that the implementation should be minimally-

[2]The intermediate representation used for this project is caled `Lambda`. It is described in more detail is Section 2.5.

invasive with respect to the underlying OCaml system it is based on. This goal is shared between the BER MetaOCaml implementation and the work presented here. As I show in Chapter 3, the resulting system is somewhere mid-way between BER MetaOCaml and "classic" MetaOCaml in terms of the changes it makes to the underlying OCaml implementation.

The contribution of this project is demonstrating the feasibility of producing a minimally-invasive implementation of MetaOCaml that uses the OCaml intermediate representation as the concrete representation of the program that is being manipulate.

After giving the necessary background on multi-stage programming and the OCaml compiler, I describe the design and implementation of the multi-stage programming extensions added to the OCaml compiler; a performance evaluation of the resulting system follows, and is followed in turn by a summary and conclusion.

Chapter 2

Background

2.1 Overview

MetaOCaml is a multi-stage programming system based on MetaML. The work presented in this thesis implements MetaOCaml as an extension to the compiler for OCaml version 4. This chapter introduces the basic notions of multi-stage programming, and gives an overview of the OCaml compiler, with particular focus on the parts of the compiler that are important for the discussion of implementation in Chapter 3.

2.2 A brief introduction to multi-stage programming

The goal of this section is to introduce readers unfamiliar with multi-stage programming to the basic concepts, and to provide some simple examples of multi-stage programs.

Multi-stage programming is an extension of ideas found in the field of partial evaluation, so, in order to explain multi-stage programming, I start by describing partial evaluation.

2.2.1 Partial evaluation

Partial evaluation may be described informally as running a program when values are known for only some of the program's variables. Naturally, if values for all of

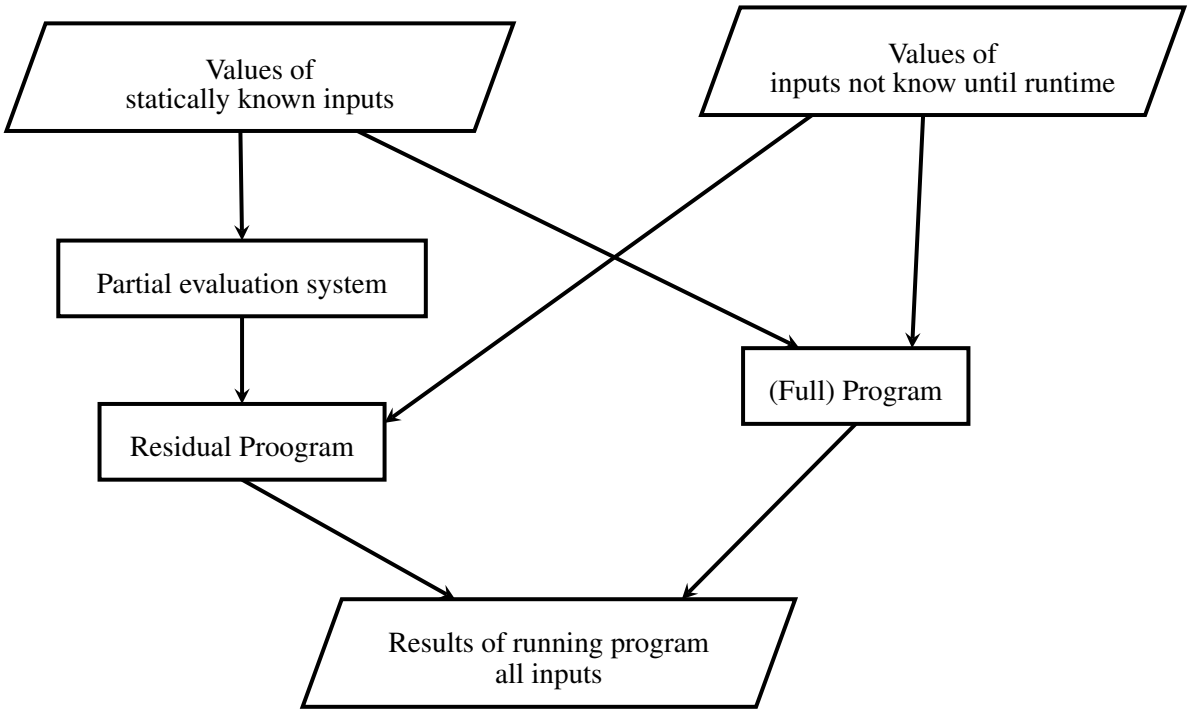

Figure 2.1: Partial evaluation data flow.

the program's variables are known, the program can just be run in the usual fashion to obtain the result. If only some of the values are known, however, then a program along with the known values may be given to a partial evaluator, which will analyze the program and the available inputs and produce a residual program, as illustrated in Figure 2.1. When the values for the variables that were not yet known at the time the partial evaluator ran become known, the residual program may be run with just those *extra* values to obtain the result. The correctness criteria for partial evaluation is quite natural: the result obtained should not depend on whether a program was run in the normal fashion, with all the values specified or whether the program was first run through the partial evaluator with only some of the values known, and then the residual program was run on the remaining values.

To describe the process of partial evaluation in more detail, I will use a simple

$$x \in \texttt{VARIABLE} \quad n \in \mathbb{N} \quad t \in \texttt{TERM} v \in \texttt{VALUE}$$
$$t ::= n \,|\, x \,|\, \lambda\, x.\, t \,|\, t\ t \,|\, \mathrm{fix}\, x.\, t \,|\, t\ ?\ t\ :\ t \,|\, t\ +\ t \,|\, t\ -\ t \,|\, t\ *\ t$$
$$v ::= n \,|\, \lambda\, x.\, t$$

Figure 2.2: Syntax of a toy language for partial evaluation

"toy" language. The syntax of the language is given in Figure 2.2.[1]

The language is typical of functional programming languages inspired by Church's lambda calculus [6]. The intuition is that writing down one of the terms (TERM) creates an *expression*; expressions may be *evaluated* to obtain their value. The rules of evaluation appear below, but the intuitive meaning of numbers (n), variables (x), and arithmetic operations on natural numbers ($+, -, *$) is probably clear to anyone familiar with at least one programming language. The conditional expression ($t\ ?\ t_t\ :\ t_f$) works like the conditional operator $t\ ?\ t_t\ :\ t_f$ found in C (and Java). Writing down $\lambda\, x.\, t$ creates a function expression. Function expressions may be applied to an argument using $t\ t$ (the rules by which application is evaluated appear below). Finally, $\mathrm{fix}\, x.\, t$ creates recursive expressions: if one writes down $\mathrm{fix}\, x.\, t$, then, within t, x stands for the entire expression $\mathrm{fix}\, x.\, t$, and $t\ t$ may be used to apply that expression to an argument. A word on precedence is in order: function application has the highest precedence, and in particular the precedence of a function application is higher than that of any of the binary operators. This means that $f\ 1\ +\ 2$ is parsed as $(f\ 1)\ +\ (2)$, rather than as $f\ (1\ +\ 2)$ As usual, parentheses may be used to override the default precedence.

Conceptually, a result is obtained by rewriting the program according to the rules that appear in Figure 2.3 until no rewrite rules apply.

The intuition is as follows: Numbers,and *abstractions* (the standard name for $\lambda\, x.\, t$, which defines an anonymous function) evaluate to themselves. To evaluate an *application* (the standard name for $t\ t$), one evaluates both the term (t_1) being applied (the *operator*) and the argument of the application (t_2) (the *operand*): the operator *must* evaluate to a $\lambda\, x.\, t$-form (however it is immaterial what the operand

[1] More formally, the BNF rule describes the set of syntactically valid terms of the language.

evaluates to); once both the operator and the operand are known, one substitutes the operand for the formal parameter of the $\lambda x.\ t$-form ($[v/x]$) in the body of the $\lambda x.\ t$-form, while being careful not to *capture* variables by accident (more details on the rules of substitution are found below); finally, one evaluates the term that results from the substitution: the value obtained from evaluation is the value of the application. The $\text{fix} x.\ t$ expression allows recursive functions to be written: the intuition is that $\text{fix} x.\ t$ gives a name to the function that is in the process of being defined, so that the name can be used inside the body of the function to allow the defintion to "refer to itself.". More formally, after writing down $\text{fix} x.\ t$, whenever x is applied to an argument, the rules of evaluation ensure that an entire new copy of $\text{fix} x.\ t$ will be substituted in place of x (to avoid infinite recursion, one needs to ensure that applying $\text{fix} x.\ t$ to an argument doesnt *always* result in an application of x to some argument). The intuition for $t\ ?\ t_t\ :\ t_f$ is that it works like the ternary ? : operator in C (and Java).

A combination of conditional and (recursive) function calls enables a wide range of flow control constructs to be expressed in the language. A few illustrations of programs written using in the toy language are forthcoming, but, to finish the presentation of the language, *capture-avoiding substitution* needs to be described.

Capture-avoiding substitution

In an expression like $\lambda x.\ x + y$, the variable x is *bound*, while the variable y is *free*, and analogously for the $\text{fix} x.\ x + y$. The function FV t produces a set of variables that are free in a term in the toy language. The formal definition of the FV t function appears in Figure 2.5. A variable is *bound* if it is not free.

The intent of the *capture-avoiding substitution* is that bound variables should stay bound following the substitution and free variables should stay free: this way, the "meaning" of terms is preserved by substitution. To demonstrate an example of a *variable capture*, if one were to perform textual substitution $(\lambda x.\ y)[y := x]$ to obtain $\lambda x.\ x$, a variable that started out free (namely, y) has been *captured* following textual substitution.

I now give a few examples to illustrate how familiar functions can be expressed

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \frac{t \to t'}{v\ t \to v\ t'} \qquad \frac{}{(\lambda\, x.\ t)\ v \to t[v/x]}$$

$$\frac{}{\mathrm{fix}\, x.\ t \to t[(\mathrm{fix}\, x.\ t)/x]}$$

$$\frac{t \to t'}{t\ ?\ t_1\ :\ t_2 \to t'\ ?\ t_1\ :\ t_2} \qquad \frac{n \neq 0}{n\ ?\ t_1\ :\ t_2 \to t_1} \qquad \frac{}{0\ ?\ t_1\ :\ t_2 \to t_2}$$

$$\frac{t_1 \to t_1'}{t_1 + t2 \to t_1' + t2} \qquad \frac{t_2 \to t_2'}{v + t_2 \to v + t_2'} \qquad \frac{n = n_1 + n_2}{n_1 + n_2 \to n}$$

$$\frac{t_1 \to t_1'}{t_1 - t2 \to t_1' - t2} \qquad \frac{t_2 \to t_2'}{v - t_2 \to v - t_2'} \qquad \frac{n = n_1 - n_2}{n_1 - n_2 \to n}$$

$$\frac{t_1 \to t_1'}{t_1 * t2 \to t_1' * t2} \qquad \frac{t_2 \to t_2'}{v * t_2 \to v * t_2'} \qquad \frac{n = n_1 * n_2}{n_1 * n_2 \to n}$$

Figure 2.3: Semantics of a toy language for partial evaluation

in this "toy" language. The familiar factorial function is

$$\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)$$

.

The function to calculate the *n*th Fibonacci number is

$$\mathrm{fix}\, f.\ \lambda\, n.\ \lambda\, a.\ n\ ?\ \lambda\, b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a$$

The evaluation of the expression

$$(\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1))\ 3$$

$$
\begin{aligned}
\mathrm{FV}\ n &= \varnothing \\
\mathrm{FV}\ x &= x \\
\mathrm{FV}\ \lambda x.\ t &= \mathrm{FV}\ t \setminus \{x\} \\
\mathrm{FV}\ (t_1\ t_2) &= \mathrm{FV}\ t_1 \cup \mathrm{FV}\ t_2 \\
\mathrm{FV}\ \mathrm{fix}\, x.\ t &= \mathrm{FV}\ t \setminus \{x\} \\
\mathrm{FV}\ (t_1\ ?\ t_2\ :\ t_3) &= \mathrm{FV}\ t_1 \cup \mathrm{FV}\ t_2 \cup \mathrm{FV}\ t_3 \\
\mathrm{FV}\ (t_1\ +\ t_2) &= \mathrm{FV}\ t_1 \cup \mathrm{FV}\ t_2 \\
\mathrm{FV}\ (t_1\ -\ t_2) &= \mathrm{FV}\ t_1 \cup \mathrm{FV}\ t_2 \\
\mathrm{FV}\ (t_1\ *\ t_2) &= \mathrm{FV}\ t_1 \cup \mathrm{FV}\ t_2
\end{aligned}
$$

Figure 2.4: The free variables function

$$
\begin{aligned}
[t/x]n &= n && \\
[t/x]y &= y && x \neq y \\
[t/x]x &= t && \\
[t/x]\lambda x.\ t &= \lambda x.\ t && \\
[t/x]\lambda y.\ t &= \lambda x'.\ [t/x][x'/y]t && \text{where } x \neq y \text{ and } x' \notin \mathrm{FV}\ \lambda y.\ t \cup \mathrm{FV}\ t \cup x \\
[t/x](t_1\ t_2) &= [t/x]t_1\ [t/x]t_2 && \\
[t/x]\mathrm{fix}\, x.\ t &= \mathrm{fix}\, x.\ t && \\
[t/x]\mathrm{fix}\, y.\ t &= \mathrm{fix}\, x'.\ [t/x][x'/y]t && \text{where } x \neq y \text{ and } x' \notin \mathrm{FV}\ \mathrm{fix}\, y.\ t \cup \mathrm{FV}\ t \cup x \\
[t/x](t_1\ ?\ t_2\ :\ t_3) &= [t/x]t_1\ ?\ [t/x]t_2\ :\ [t/x]t_3 && \\
[t/x](t_1\ +\ t_2) &= [t/x]t_1\ +\ [t/x]t_2 && \\
[t/x](t_1\ -\ t_2) &= [t/x]t_1\ -\ [t/x]t_2 && \\
[t/x](t_1\ *\ t_2) &= [t/x]t_1\ *\ [t/x]t_2 &&
\end{aligned}
$$

Figure 2.5: The capture-avoiding substitution function

$$
\begin{aligned}
&(\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1))\ 3 = \\
&(\lambda\, n.\ n\ ?\ n * (\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1))(n-1)\ :\ 1)\ 3 = \\
&3\ ?\ 3 * \mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ (3-1)\ :\ 1 = \\
&3 * (\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ 2) = \\
&3 * ((\lambda\, n.\ n\ ?\ n * \mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ (n-1)\ :\ 1)\ 2) = \\
&3 * (2\ ?\ 2 * \mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ (2-1)\ :\ 1) = \\
&3 * 2 * (\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ 1) = \\
&3 * 2 * ((\lambda\, n.\ n\ ?\ n * \mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ (n-1)\ :\ 1)\ 1) = \\
&3 * 2 * (1\ ?\ 1 * \mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ (1-1)\ :\ 1) = \\
&3 * 2 * 1 * (\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ 0) = \\
&3 * 2 * 1 * ((\lambda\, n.\ n\ ?\ n * \mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ (n-1)\ :\ 1)\ 0) = \\
&3 * 2 * 1 * (0\ ?\ 0 * \mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)\ (n-1)\ :\ 1) = \\
&3 * 2 * 1 * 1 = \\
&6
\end{aligned}
$$

Figure 2.6: The evaluation of $(\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1))\ 3$

proceeds as shown in Figure 2.6

I can now give an example of partial evaluation. If I wish to obtain a function for calculating *n*th Fibonacci number with the value of n fixed ahead of time (e.g., $n = 3$), then the function can be obtained simply by rewriting the application

$$(\mathrm{fix}\, f.\ \lambda\, n.\ \lambda\, a.\ n\ ?\ \lambda\, b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3$$

until all occurrences of the variable n are eliminated from the expression, as shown in Figure 2.7 (to make the calculations less cluttered, I only substitute for f when the next step of the calculation depends on its definition). A noticeable difference between the partial evaluation of

$$(\mathrm{fix}\, f.\ \lambda\, n.\ \lambda\, a.\ n\ ?\ \lambda\, b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3$$

$$
\begin{aligned}
(\mathrm{fix}\, f.\ \lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3 &= \\
(\lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3 &= \\
\lambda a.\ \lambda b.\ ((f\ 3-1)\ b)\ (a+b) &= \\
\lambda a.\ \lambda b.\ (((\mathrm{fix}\, f.\ \lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 2)\ b)\ (a+b) &= \\
\lambda a.\ \lambda b.\ (((\lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 2)\ b)\ (a+b) &= \\
\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ (((\mathrm{fix}\, f.\ \lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 1)\ b)\ (a+b))\ b)\ (a+b) &= \\
\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ (((\lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 1)\ b)\ (a+b))\ b)\ (a+b) &= \\
\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ ((f\ 1-1)\ b)\ (a+b))\ b)\ (a+b))\ b)\ (a+b) &= \\
\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ ((f\ 0)\ b)\ (a+b))\ b)\ (a+b))\ b)\ (a+b) &= \\
\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ ((\lambda a.\ a)\ b)\ (a+b))\ b)\ (a+b))\ b)\ (a+b) &
\end{aligned}
$$

Figure 2.7: Partial evaluation of $(\mathrm{fix}\, f.\ \lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3$

and the standard evaluation of

$$(\mathrm{fix}\, f.\ (\lambda n.\ n\ ?\ n*f\ (n-1)\ :\ 1))\ 3$$

is that, during the evaluation of

$$(\mathrm{fix}\, f.\ (\lambda n.\ n\ ?\ n*f\ (n-1)\ :\ 1))\ 3,$$

the work of calculation (*reduction*) always happened at the outer-most level of the term (a pattern typical of *tail-recursive* functions, which are equivalent to **while** loops). On the other hand, during the partial evaluation of

$$(\mathrm{fix}\, f.\ \lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3,$$

the reduction happened *inside* the term, in a pattern typical of functions that are not tail recursive – even though, as written, the function

$$\mathrm{fix}\, f.\ \lambda n.\ \lambda a.\ n\ ?\ \lambda b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a$$

is tail-recursive when fully applied.

While the partial evaluation of

$$(\text{fix}\, f.\, \lambda\, n.\, \lambda\, a.\, n\, ?\, \lambda\, b.\, ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3$$

may demonstrate the process of partial evaluation, it leaves the question of how a partial evaluator program might be implemented somewhat vague. To understand multi-stage programming, a high-level understanding of how a partial evaluator works is helpful, so I will sketch out the architecture of a partial evaluation system.

2.3 Partial evaluation systems

A typical partial evaluation system [18] can be thought of as occurring in two distinct phases: a *binding time analysis* phase which annotates the source code to indicate where computation can be done using the statically known inputs, and the specialization phase, which transforms the parts of the program annotated by the binding time analyzer to produce a *residual program*. The residual program itself can then be subjected to partial evaluation, producing in turn a new residual program, which could then be partially evaluated. As a simple example of this process of "chained" partial evaluations, the residual program obtained in Figure 2.7 could be partially evaluated if the value of the starting term of the Fibonacci sequence was supplied. I have found it easiest to understand multi-stage programming as building on the notion of "chained" partial evaluation.

2.4 Multi-stage programming in MetaOCaml

The MetaOCaml multi-stage programming system does not include an automated binding-time analysis phase: instead, it requires the programmer to perform binding-time analysis manually, and to *annotate* the source code with *staging annotations* (which are essentially binding-time annotations). Also, in addition to the usual values (such as integers, strings, etc.), a MetaOCaml program may produce code for a program that contains staging annotations (i.e., asking a MetaOCaml system to evaluate suitably annotated

$$(\text{fix}\, f.\, \lambda\, n.\, \lambda\, a.\, n\, ?\, \lambda\, b.\, ((f\ (n-1))\ b)\ (a+b)\ :\ a)\ 3$$

Table 2.1: Translation from λ-calculus to (Meta)OCaml syntax

λ-calculus syntax	OCaml syntax
x	x
n	n [3]
$\lambda\, x.\, t$	**fun** $x \rightarrow t$ [4]
$t_1\ t_2$	$t_1\ t_2$
$\mathrm{fix}\, f.\, t$	**let rec** $f = t$ [5]
$t_1\ ?\ t_2\ :\ t_3$	**if** t_1 **!= 0 then** t_2 **else** t_3
$t_1\ +\ t_2$	$t_1\ +\ t_2$
$t_1\ -\ t_2$	$t_1\ -\ t_2$
$t_1\ *\ t_2$	$t_1\ *\ t_2$

will produce a value that looks very much like the final line of Figure 2.7 – except, of course, expressed in MetaOCaml, rather than λ-calculus syntax).

To illustrate, I will show below how to annotate the Fibonacci function from above in MetaOCaml. A translation from the λ-calculus syntax which I have been using so far to (Meta)OCaml syntax is given in Table 2.1[2].

So, the definition of

$$\mathrm{fix}\, f.\ (\lambda\, n.\ n\ ?\ n * f\ (n-1)\ :\ 1)$$

becomes

$$\textbf{let rec}\ f\ =\ (\textbf{fun}\ n\ \rightarrow\ \textbf{if}\ n\ \textbf{!= 0 then}\ n * f\ (n-1)\ \textbf{else}\ 1),$$

and the definition of

$$\mathrm{fix}\, f.\ \lambda\, n.\ \lambda\, a.\ n\ ?\ \lambda\, b.\ ((f\ (n-1))\ b)\ (a+b)\ :\ a$$

[2]The translation given in the table is intended to produce OCaml code that is as close to the corresponding λ-calculus syntax as possible (as opposed to producing idiomatic OCaml)

[3]Negative integers need to be written as $\tilde{}-1$, $\tilde{}-2$, etc. in OCaml syntax

[4]The arrow($\rightarrow$) is typed as a sequence of two characters: **->**

[5]**let rec** $f = t$ **in** needs to be used if **let rec** $f = t$ appears within another term (as opposed to at the outer level of a term), e.g., , **let rec** $f =$ **fun** $x \rightarrow t \ldots$, but **fun** $x \rightarrow$ **let rec** $f =$ **fun** $y \rightarrow \ldots$ **in** $\ldots$

becomes

let rec f = **fun** $n \rightarrow$ **fun** $a \rightarrow$ **if** n **!= 0 then fun** $b \rightarrow ((f\ (n-1))\ b)\ (a+b)$ **else** a.

In MetaOCaml, three kinds of annotations are used to express the order of staging (or, equivalently, the result of binding-time analysis). The code brackets `.<,>.` mark a term as *code* (i.e, not evaluated); the unary operator `.~` (`escape`) is used inside a code block to request that an expression be evaluated, and a result be spliced into a piece of code, and, finally `.!` (`run`) is used to run a piece of code. As a practical matter, just `code` and `escape` are sufficient for the annotations most programmers would want to write most of the time. A key property of the design of MetaOCaml system is that it is always possible to erase all annotations from a piece of MetaOCam code and end up with valid code in plain OCaml.

Taha [30] reports that, in his experience, annotating functions as code transformers (i.e., functions take arguments, some of which may be pieces of code) tends to produce fewer annotations than annotating functions as "pieces of code that happen to contain functions". With that in mind, I start the annotation of the function to calculate the nth Fibonacci number. Here is the original function again:

let rec f = **fun** $n \rightarrow$ **fun** $a \rightarrow$ **if** n **!= 0 then fun** $b \rightarrow ((f\ (n-1))\ b)\ (a+b)$ **else** a

To make the definition of the function legal under the OCaml type system, which requires that both branches of the `if` conditional produce result of the same type, I rearrange the definition by "moving out" the **fun** $b \rightarrow$

let rec f = **fun** $n \rightarrow$ **fun** $a \rightarrow$ **fun** $b \rightarrow$ **if** n **!= 0 then** $((f\ (n-1))\ b)\ (a+b)$ **else** a

Borrowing inspiration from Roberts [26] and Felleisen *et al.* [9], I can contrast the process of writing regular Fibonacci numbers function with the process of adding staging annotations to the Fibonacci numbers function as follows: when writing regular Fibonacci numbers function, *the recursive leap of faith* [26] is that the recursive call to the function produces a previous Fibonacci number in the

sequence; the job of the programmer is to use operations on numbers (addition, subtraction, multiplication, etc.) to construct an expression for the next Fibonacci number given the recursive leap of faith [9]. On the other hand, when adding multi-stage annotations to the Fibonacci numbers function, the recursive leap of faith is that the call to the function produces *the code which calculates* the previous Fibonacci number. The job of the multi-stage programmer is to use operations on code (i.e., quoting using meta-brackets and splicing using `.~` to produce the code which calculates the next Fibonacci number in the sequence, given the recursive leap of faith.

Both the unannotated and the annotated version of the Fibonacci function take three parameters, but the *meanings* of the parameters differ: while the unannotated version takes a count of how many numbers lie between the parameter named a and the desired Fibonacci number, the value of the second-most-recent Fibonacci number calculated so far, and the value of the most-recent Fibonacci number calculated so far (in that order), in the annotated version, the meaning of the first parameter is unchanged, but the second and third parameter are now pieces of code which calculate the respective Fibonacci numbers. Also, the return values of the two versions of the function differ: the unannotated version of the function returns the desired Fibonacci number; the annotated version returns the code which calculates the desired Fibonacci number.

What annotations need to be added to the function? For the $n = 0$ branch of the conditional, if the function is given a piece of code that calculates the desired Fibonacci number, it simply returns that piece of code. On the other hand, in the $n \neq 0$ case, a holds a piece of code which calculates $i-1$st Fibonacci number, while b holds a piece of code which calculates ith Fibonacci number. The function needs to construct and pass to the recursive call a piece of code which calculates the $i+1$st Fibonacci number. A template for the piece of code to be constructed can be written down as `.<...+ ...>.`. The first idea might be to write down simply `.<a + b>.`; however that expression is rejected by MetaOCaml. The reason the expression is rejected is that it is *nonsense*[8]: plus($+$) operates on numbers, not on pieces of code. The MetaOCaml `.~` operator takes a piece of code that calculates x, and splices x into a fragment of code that surrounds the application of the operator. The final annotated function then, is:

```
let rec f = fun n ->
    fun a ->
        fun b -> if n != 0 then ((f (n-1) b) .<(.~a+.~b)>.
                           else a
```

The behavior of the annotated function, when fully applied (i.e., when n, a and b have all been supplied) is indistinguishable from the original function. However, the compiler has been directed how to produce a partial function when the function is *partially* applied to just n (or just n and a).

2.5 OCaml compiler overview

The goal of this project was to re-implement MetaOCaml on top of a modern OCaml compiler, targeting the native code generation interface of the OCaml compiler. Before describing the specifics of the implementation strategies chosen, I describe the structure of the OCaml compiler to help the reader orient herself.

At a high level, the OCaml compiler is structured similarly to other modern compilers, as shown in Figure 2.8: there is a compiler front end which produces the intermediate representation of the program's code and several back-ends which work on the intermediate representation to generate code for various target instruction sets.

The front end that is part of the standard OCaml distribution is responsible for performing lexical analysis and parsing of the OCaml source code as well as for type checking the program, as illustrated in Figure 2.9. A key data structure is an abstract syntax tree (abstract syntax tree (AST)), as defined in the file `parsing/parsetree.mli`[6]. The standard OCaml distribution also includes a preprocessor which allows ASTs to be transformed before they are handed off to the type checker.

The OCaml type checker works on ASTs, and produces ASTs that are annotated with typing annotations. These typed ASTs are defined in the file `typing/typedtree.mli`. The type checking is the final stage at which a program can be rejected: once an OCaml program is accepted by the type checker, it *will* be compiled, producing an executable for a target architecture. Further, the OCaml

[6] All paths in this chapter are rooted at the `src` directory of the standard OCaml distribution

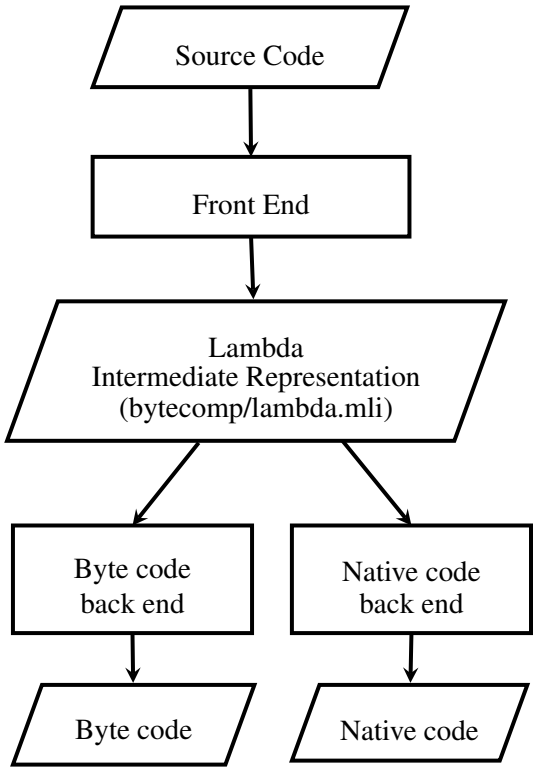

Figure 2.8: OCaml compiler overview.

type system guarantees that a program that is accepted by the type checker will be free from certain kinds of errors at run time.[7]

Once the AST has been successfully type-checked, the subsequent stages of compilation do not use typing information.[8]

A key data structure for the intermediate representation is defined in the file `bytecomp/lambda.mli`. The location of the files `lambda.mli` and `translcore.mli` in the source tree notwithstanding, *both* the byte code back

[7] In theory. In practice, the type checker is just a program which could – in theory – have bugs in it. Outside of the somewhat esoteric possibility of bugs in the OCaml type checker, OCaml includes a suite of unsafe operations in the module `Obj` – including an operation equivalent to the C++ reinterpret_cast (namely, Obj.magic): since (some) of those operations subvert the OCaml type system, if a program uses one of those operations, then – generally speaking – all bets are off at run time.

[8] This "throwing away" of type information is characteristic of not only OCaml but most compilers designed in the 90s [7]. Some of the compilers developed from 2000 onward retain type information much later in the compilation process [32].

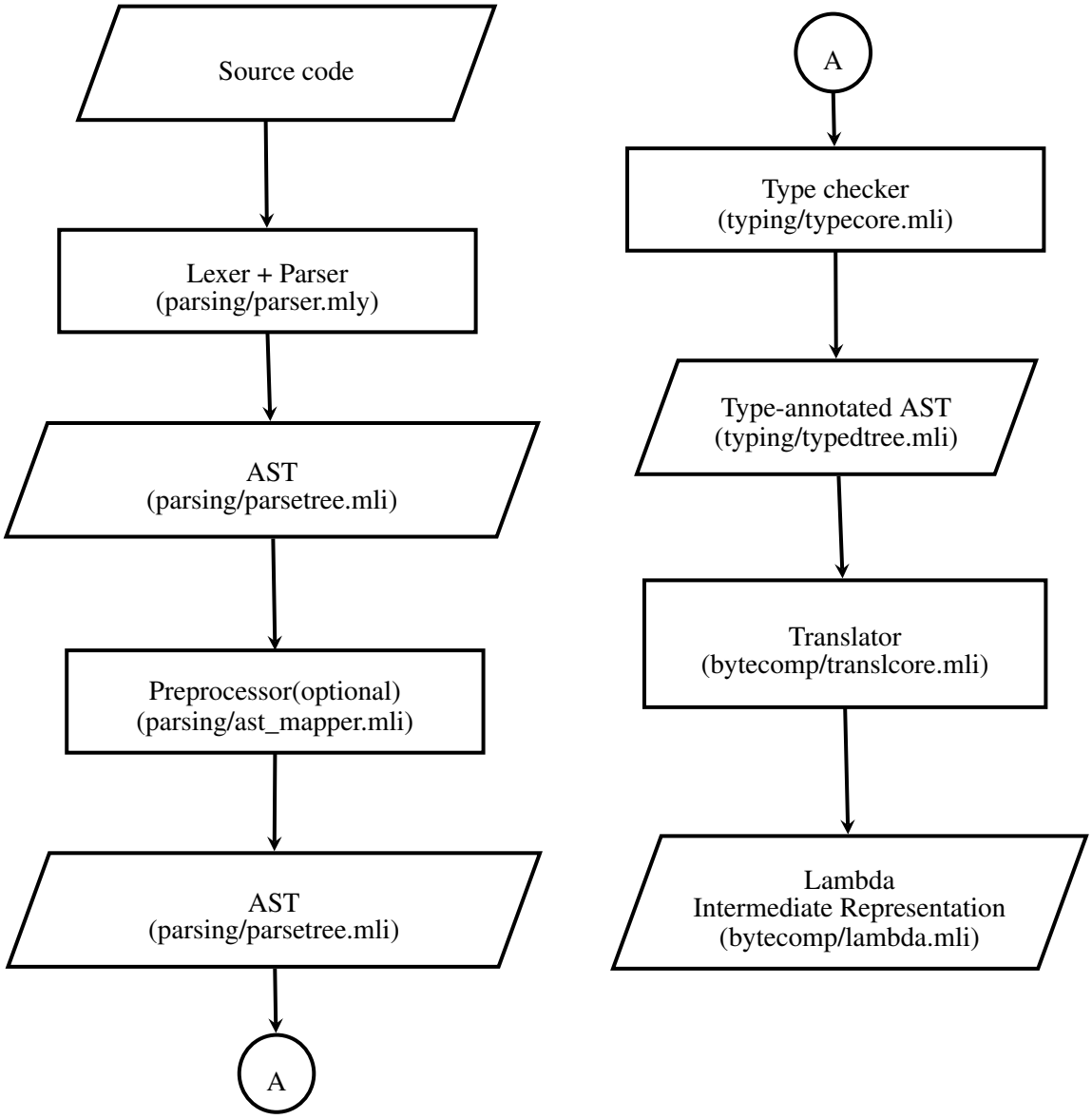

Figure 2.9: OCaml compiler frontend detail.

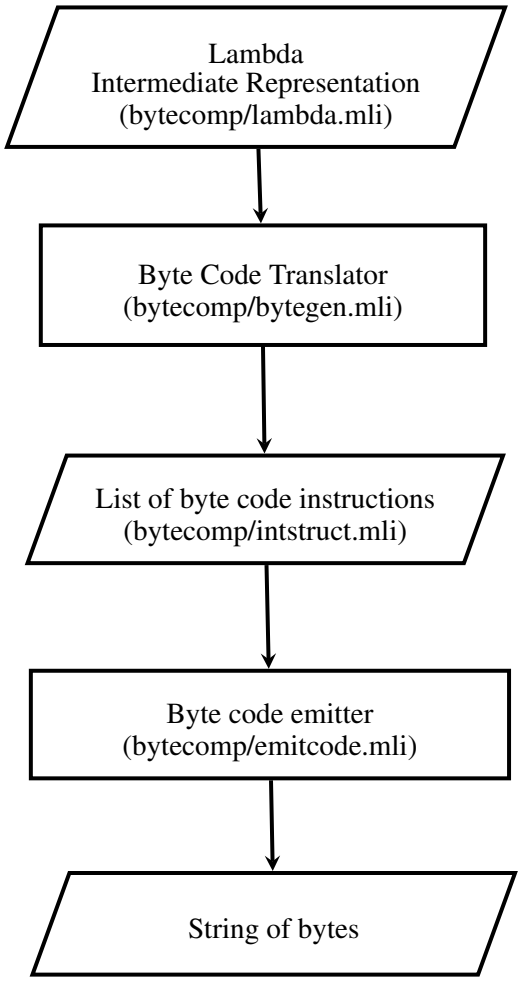

Figure 2.10: OCaml bytecode backend detail.

end and the native code back-end work on the data types defined in `lambda.mli`, so, at this point in the development of the OCaml compiler it is quite reasonable to think of Lambda as *the* intermediate representation in the OCaml compiler.

Type-annotated ASTs are translated to the Lambda intermediate representation by the functions declared in the file `bytecomp/translcore.mli`

The byte code back end, defined in the file `bytecomp/bytegen.mli`, works on the lambda representation to produce a stream of byte code instructions (`bytecomp/instruct.mli`) for the OCaml byte code interpreter, as shown in

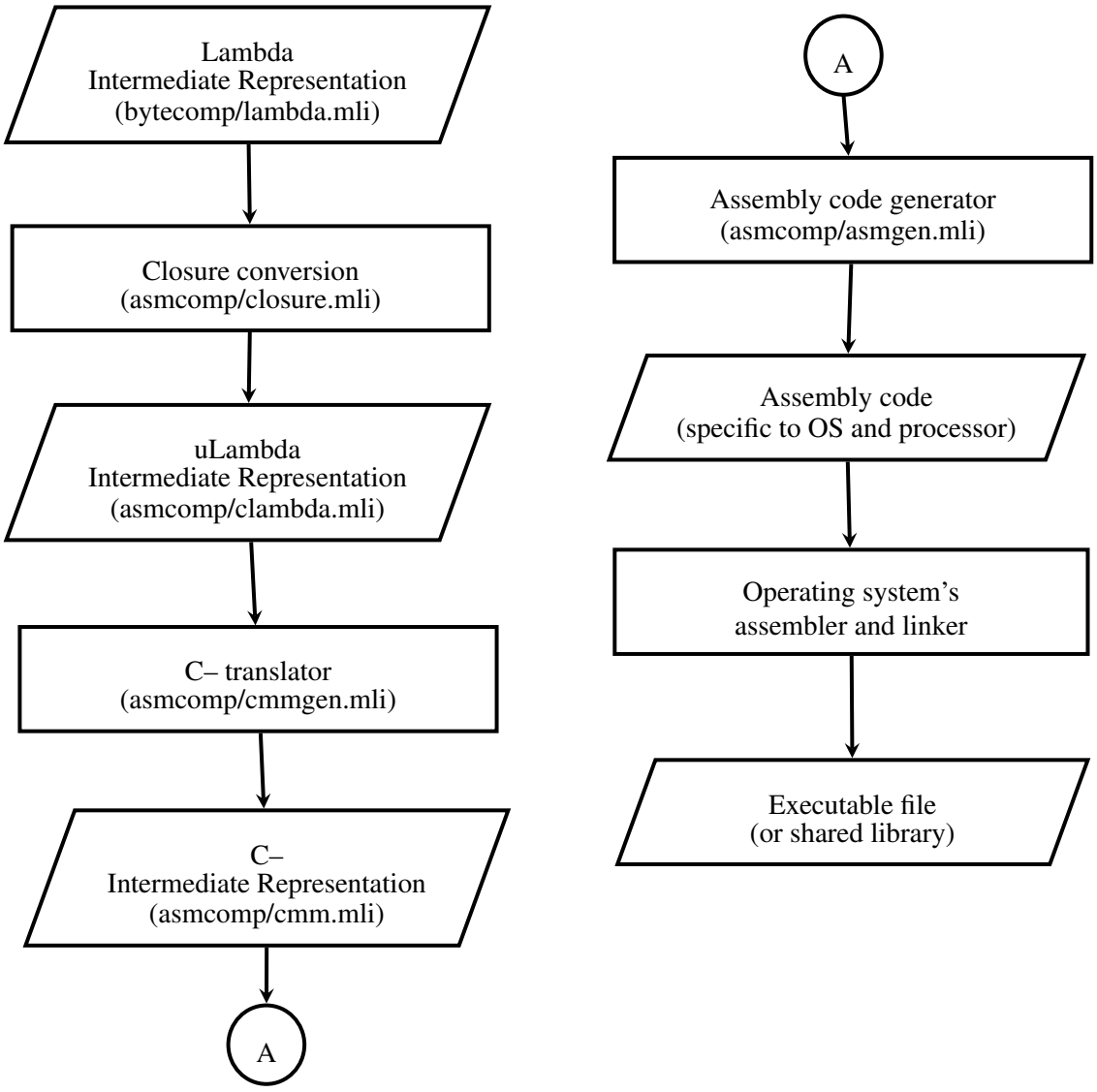

Figure 2.11: OCaml native code backend detail.

Figure 2.10. Ultimately, a string of bytes may be emitted either to memory or to a file on disk.

The native code back end has a few more moving parts than the byte code back end, as shown in Figure 2.11. The intermediate representation is first closure-

converted [3] and a bit of data flow analysis is done on it by the function intro declared in the file `asmcomp/closure.mli`, producing a representation called μlambda (`asmcomp/clambda.mli`). The μlambda representation is then converted to the C-- language by `asmcomp/cmmgen.mli` (Incidentally, the C-- language in the OCaml compiler is not related [24] to the C-- language of Simon Peyton Jones [25]). Assembly code for a specific processor (e.g, i386, x86_64, arm, etc.) is generated from the C-- representation via a series of standard code generation operations, such as register allocation, pipeline scheduling, etc. The generated assembly code is then handed off to the system's assembler and linker (the latter produce executables in the system's preferred object file format).

2.5.1 OCaml values

The OCaml run-time system is garbage-collected, so, similarly to other garbage-collected run-time systems [13] [12] OCaml makes a distinction between *immediate* and *pointer* values. Broadly speaking, immediate values are values whose run-time representation is small enough that it is practical to pass the value on the C-like run-time stack. On the other hand, pointer values, *point to* (i.e., hold the address of) a garbage-collected value on the heap. A language implementer makes a choice about how small is "small enough" and what is "practical" to pass on the run-time stack. Given a value, an implementation needs to be able to distinguish at run time whether the value is immediate or a pointer. To this end, the common implementation strategy is to *tag* values: the implementer decides that a few bits of a value (usually starting with the least significant bit) are to be used to distinguish pointers from immediate values. The usual trick of the trade is to observe that every microprocessor platform currently in common use, either requires that pointer values be *aligned* (i.e., hold an address that is a multiple of a power of 2) or strongly encourages aligned pointer access (by providing more efficient access to aligned as opposed to *unaligned* pointers, the latter being pointer values that are not a power of 2). That observation leads to an implementation decision to only use aligned pointers into the implementation's run-time heap. Since an aligned pointer holds an address that is a power of 2, the low bits of such a pointer will always be 0 (exactly how many bits depends on the minimum alignment chosen). This means

that, if a value does not have 0 in the low bit(s) it *cannot* be a pointer, and must therefore be an immediate value.

The choices made by CamlLight (a byte-code-only precursor to OCaml) were described by Leroy [22]. That description is still surprisingly accurate when it comes to describing how values are laid out in the OCaml heap. The choices OCaml makes are actually quite simple: the only immediate values are integers, so only one tag bit is needed. Hence, pointers can (in principle) be aligned on 2-byte boundaries, and on a machine with 32-bit integers, immediate integers can range from $2^{31} - 1$ to 2^{31}.

OCaml closures

The implementation of MetaOCaml `code` relies crucially on notions of a function's environment and closure, so it is useful to describe those notions in some detail. The material in this section forms the basis for the presentation of the implementation in Section 3.2.

A function's *environment* maps names of variables which are free 2.2.1 in the function to the values associated with those variables. A *closure* [21] is a data structure that combines the code of the function with the function's environment.

A closure is a key mechanism for implementing *lexical scope* in presence of *first-class functions*. A (type of) value in a programming language is said to be *first class* if the language supports performing all of the following operations on an object:

- Passing the value as an argument to functions
- Returning the value as a result of a function
- Storing the value in a data structure and assigning values to variables

To illustrate the need for a closure I invite the reader to consider the example function that appears in Figure 2.12[9]. The function `make_adder` returns a function which adds *n* to its argument. Lexical scoping rules demand that the `n` in the body of `adder` should refer to the value that was passed as an argument in

[9]The function is written in a fictional programming language

```
function make_adder(n):
    function adder(x):
        return x+n
    return adder
```

Figure 2.12: A function that returns a function

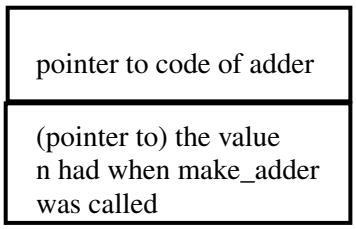

Figure 2.13: The memory of the closure created for adder

the call to `make_adder`. Another way to say the same thing is that the environment associated with `adder` maps the name `n` with the value `n` had at the time `make_adder` was called. In order to support lexical scoping rules, the compiler needs to "remember" not just the code of the function `adder`, but also the value that `n` had when `make_adder` was called. Because of this need to "remember" not just the code of the function but also its environment, the compiler cannot represent the function `adder` with just a pointer to the function's code: if it did, it would lose information about `adder`'s environment. One possible solution to that problem is to represent `adder` with an in-memory structure shown in Figure 2.13. To arrive at that structure, the compiler might go through the following steps:

1. Make a list of all the variables that are free in the function `adder`. In this example, the only free variable is `n`.

2. Calculate how much space is needed in order to store information about the `adder` function. In the current example, the compiler is only storing a pointer to the code of the function, but more information could be kept[10].

3. For each free variable, calculate the offset at which the value the variable refers to will be stored with respect to the beginning of the in-memory struc-

[10]For example, the OCaml compiler stores (at least) the number of parameters a function takes in addition to a pointer to the function

ture that is being created. In this example, there is only one variable named `n`, so the compiler would "remember" that "variable `n` can be found at (zero-based) offset 1 in the memory block". In general, more than one variable may be free, and a table mapping names to offsets may need to be built at compile time.

4. Add a parameter which holds a pointer to the function's environment to the list of parameters for the function, and replace references to names in the body of the function with offsets from the environment parameter, according to the table created in the previous step. In the current example, the compiler would arrange for `adder` to take an extra parameter that holds a pointer to its environment, and, in the body of `adder`, the name `n` would get replaced with an offset (in this case, 1) from the environment pointer.

Having completed these steps, the compiler can pass around a pointer to the in-memory structure in Figure 2.13 in lieu of a code to just the code of `adder`. The process of creating this in-memory structure is known as *closure conversion* [3].

To compile a call to `adder` under the assumption that the compiler "has in its hands" a pointer to the structure shown in Figure 2.13 (which I will refer to simply as *closure*), the compiler proceeds as follows:

1. Retrieves a pointer to the function from the pointer to the closure. In the current example, the pointer to the function is simply the pointer stored at offset zero with respect to the the pointer to the closure.

2. Arranges for the arguments that appear as part of the call to be passed to the function (for example, the arguments may be pushed onto the stack). In addition to passing each of the arguments to the call, pass in a pointer to the closure as the value of the environment parameter.

3. Generate a jump to the function's code.

The OCmal compiler uses *flat closures* [5]. The basic idea is that a closure block is allocated that holds function information for the entire set of mutually-recursive functions, as shown in the example that follows.

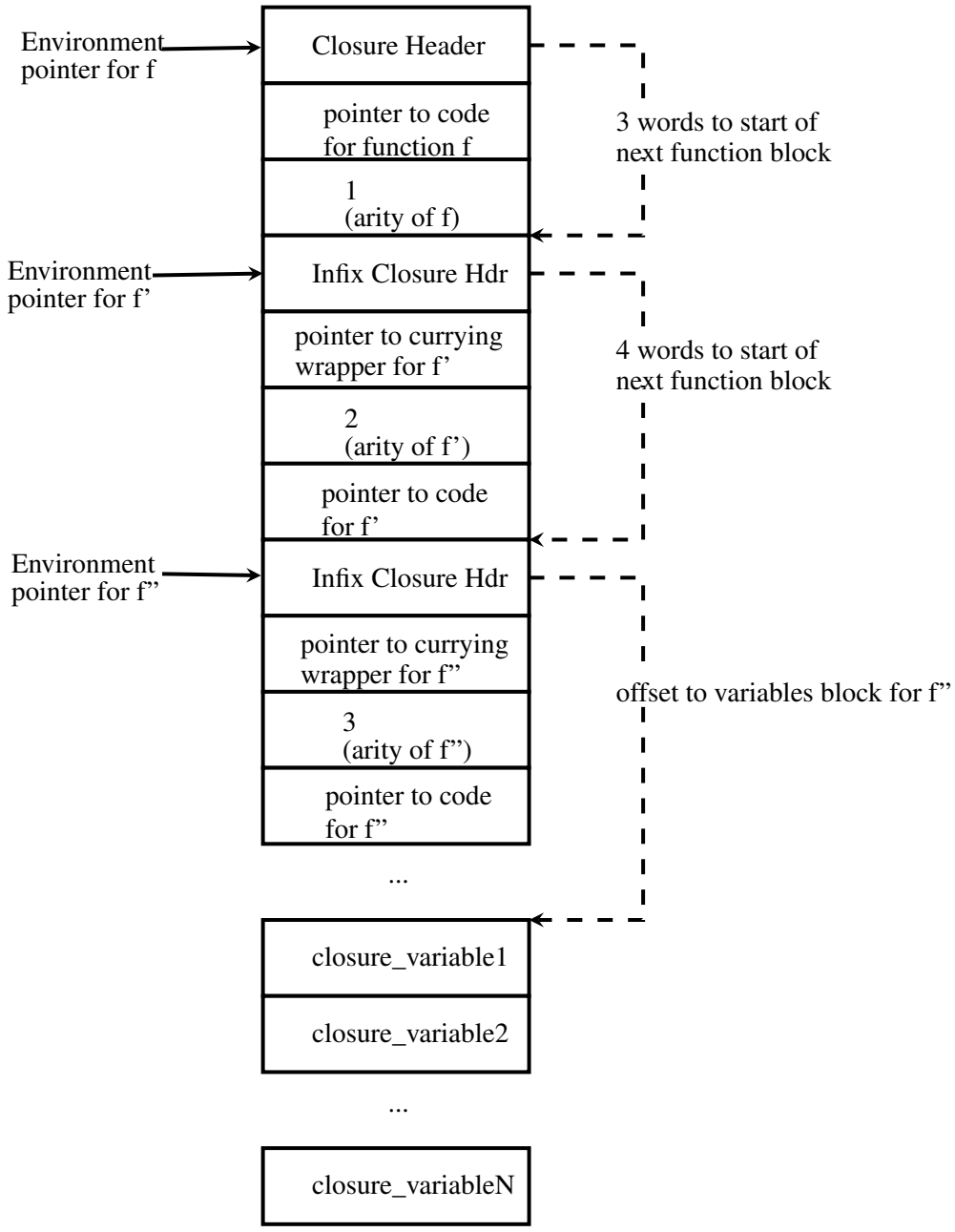

Figure 2.14: The memory layout of a standard OCaml closure

The closure shown in Figure 2.14 represents a closure the OCaml compiler would create if it encountered a definition of three mutually-recursive functions named `f`, `f'`, and `f''`; the function f takes a single argument (its *arity* is 1); the function f' takes two arguments (its arity is 2); the function f'' takes 3 arguments. The closure starts with a closure header[11]. The header is followed by one or more function blocks (in this case, there are three function blocks: one each for the functions `f`, `f'`, and `f''`. The second field of each function's block is always the arity of the function for which the closure block is being allocated. The contents of the first field depends on the function's arity: for functions of arity 1, the first field holds the pointer to the function's code; for functions of arity greater than 1, the first field holds the pointer to a *currying* [27][12] wrapper for the original function, and the third field contains the pointer to code for the function. Second and subsequent function blocks are preceeded by an infix closure header (all values in the OCaml heap are preceeded by a header; closures that appear within a flat-closure block are values, so they need to be preceeded by a header). The variables for the entire set of mutually-recursive functions are laid out in a single contiguous memory block. When compiling the definitions of the functions used in this example, the steps the OCaml compiler takes to create the closure structure can be described as follows:

1. Make a list of all the mutually-recursive functions that appear in a single mutually-recursive function definition (in this case, the list contains three functions: `f`, `f'`, and `f''`).

2. Make a list of the names of all variables that are free in all the mutually-recursive functions (in this case, there are N variable names)[13]

[11] Every value in the OCaml run-time heap has a value header, which indicates at run-time what kind of value is stored in the block. The header is also used by the garbage collector to keep track of which values are in use

[12] Currying is the process by which a function of multiple arguments is converted to a function that consumes the first argument and produces a function that consumes the rest of the arguments in the arguments list. The function that consumes the rest of the arguments in the list may in turn be curried.

[13] Name clashes do not occur because the compiler generates unique name for each variable; i.e., even if `f` and `f'` take an argument named `a`, the compiler renames the variable uniquely. To give unique name to each variable, the compiler keeps a counter which it increments each time a new variable name is encountered. The internal name of a variable is then formed by concatenating the variable the programmer wrote with the value of the counter.

3. Go through the list of mutually-recursive functions and calculate the size of the memory block needed to hold the function blocks.

4. Go through the list of names of free variables, and calculate the offset of each variable with respect to the start of the closure block. The offset of the first variable is simply the size of the memory area needed to hold the function blocks (calculated in the previous step). Subsequent variables are just laid out sequentially in the order in which they appear in the list of names of all free variables. The result of this step is a compile-time table mapping names of variables that are free in the body of the function(s) to offsets in the closure heap block. I will refer to this table as a *(compile-time) lexical environment*.

5. For each of the mutually-recursive function whose code references either a free variable or another mutually recursive function:

   - Introduce a parameter to hold the environment pointer; at function application time, the value of the environment pointer is simply the address of the start of the respective function block
   - For each of the free variables referenced, replace the variable name with an offset from the environment pointer to the location of the variable in the variables block[14].
   - For each of the mutually recursive functions referenced, replace the name of the function with an offset from the environment pointer to the location of the function in the closure block[15].

2.5.2 The OCaml toplevel

In addition to the OCaml compiler, the OCaml distribution includes an interactive *toplevel*, which operates similarly to the read-evaluate-print loop in Lisp systems[23]. For the purposes of implementing a multi-stage programming system,

[14] Because of the way OCaml compiler lays out the closure block, that offset is always a positive integer

[15] That offset may be either positive or negative, for example, the offset from the start of `f'`'s environment would be positive if `f''` is being accessed but negative if `f` is being accessed

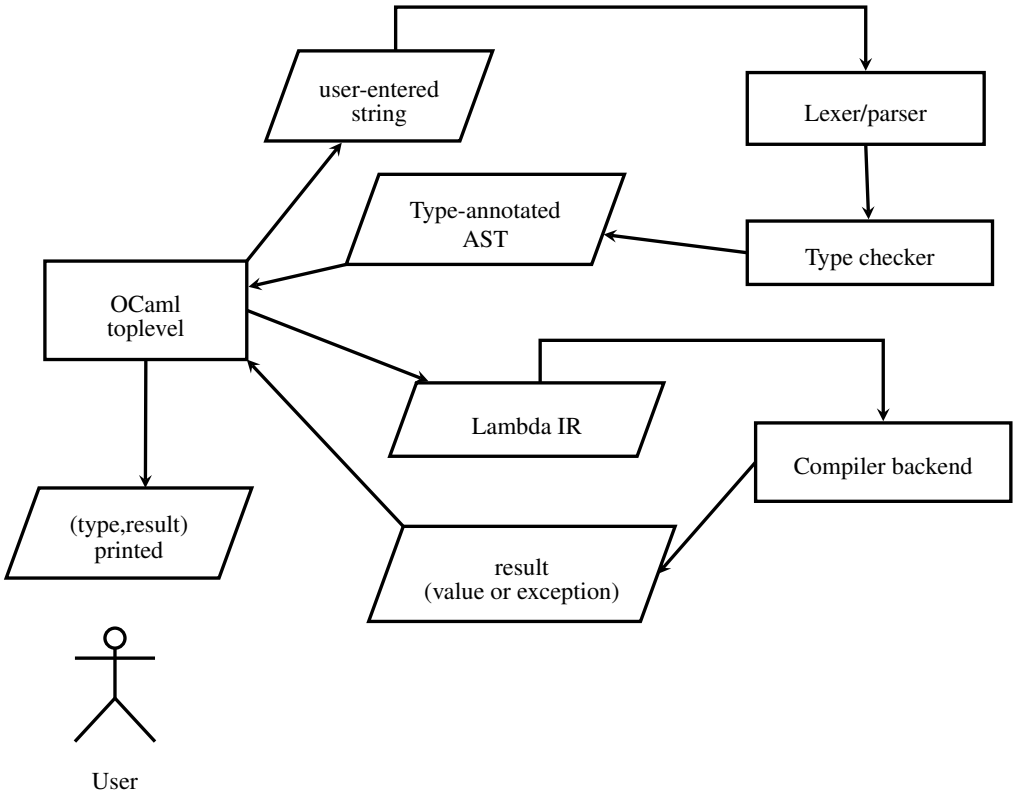

Figure 2.15: The OCaml toplevel

the toplevel is of interest because it is an interactive program that, during its execution, invokes the OCaml compiler to obtain OCaml values. This ability to call the compiler at run-time is crucial to being able to implement a multi-stage programming system, as will be seen in Section 3.1. In this section, I describe the operation and implementation of the toplevel, as it relates to my MetaOCaml implementation.

When the toplevel is started, it presents the user with a prompt at which OCaml expressions may be entered. An expression entered by the user is terminated by entering a double semicolon `;;`. The expression is then compiled and type-checked almost as it would be compiled with the regular compiler.[16] Then the expression is

[16]There is a small number of special hooks in the compiler that are used specifically to support evaluating expressions at the top level.

executed to obtain a result which can be either an exception or a regular value; the result along with the result's type are then printed to the user, and the cycle repeats.

As discussed in Section 2.5, the OCaml compiler does not pass on typing information to any compilation stages that follow the conversion to the lambda intermediate representation. Yet, as mentioned above, the toplevel prints out to the user both the value to which the expression she entered evaluates to *and* the type of the value. Clearly, the toplevel cannot obtain the type of the value directly from the compiler backend, so where does the type information come from?

The answer is that the toplevel evaluates the expression the user has entered in two distinct phases, as shown in Figure 2.15: first, it calls the type checker on the parsed expression and saves the type obtained from the type checker; then it translates the expression to the intermediate representation, and compiles and evaluates that intermediate representation to obtain a value. In the toplevel code, the function that compiles and evaluates the lambda intermediate representation is called load_lambda. As discussed in Section 2.5.3, the implementation of MetaOCaml presented in this work uses the lambda intermediate representation to represent the `code` objects. Hence, as described in more detail in Chapter 3, the MetaOCaml implementation presented here uses load_lambda as a run-time interface to the OCaml compiler.

2.5.3 Representing `code`

As was seen in Section 2.4, the central object of interest for multi-stage programming is `code`. An implementer of metaprogramming system on top of OCaml is faced with the question of which of the OCaml program representations should `code` correspond to. Choosing ASTs as the internal representation of `code` is an attractive option for several reasons:

1. Since AST is essentially a parse tree it is easy to "reconstitute" the code for display to the programmer. Since the ability to interactively explore staging annotations was one of the goals of the MetaML (and MetaOCaml) systems, the ability to display code to the programmer is important.

2. It may be possible to take advantage of the OCaml preprocessor to do some of code generation.

3. The type checking performed by the OCaml compiler on the ASTs may help catch errors in code generation.

On the other hand, using ASTs as the representation for `code` commits one to running the type checker whenever code is run – even though type systems which would reject potentially unsafe code exist [19].

Avoiding redundant type-checking was one of the goalds of this project, so I chose to use the Lambda intermediate representation, rather than ASTs as the representation for `code`. Since Lambda intermediate representation is a representation to which fully type-checked code has been translated, this approach to implementation holds the promise of reducing the amount of type checking, but leaves open the issue of how to present generated code to the programmer: while the Lambda language is quite rich, it may not be possible to unambiguously reverse-translate it into an AST.

Since all of my work was done between the intermediate representation Lambda and Cmm levels, these levels are the focus of discussion in the Chapter 3.

Chapter 3

Implementation

In this chapter, I describe my re-implementation of the MetaOCaml system. I take a chronological approach to describing the implementation, as I believe this style of presentation makes it easier to understand the design decisions and refinements that were made as the implementation progressed. Since the main object of interest is the native code compiler, the discussion focuses on it.

3.1 Run for closed terms

As a first subgoal of the project, I extended the OCaml compiler to support the code and run operations, with the limitation that code terms supported needed to be *closed* (i.e., could not contain free variables).

The compiler operation was extended as follows:

1. Support for parsing the code brackets (. $<$, $>$.) and the run operator (.!) was added) to the parser (`parsing/parser.mly`.

2. A new built-in type `code` was added (in `typing/predef.mli`).

3. The lambda intermediate representation was extended to include a type Lcode. Parsing an OCaml expression surrounded by the code brackets (. $<$, $>$.) produces the Lcode intermediate representation.

4. In the compiler backend, the lambda intermediate representation of the expression that appeared between the code brackets code brackets (. $<$, $>$.) is

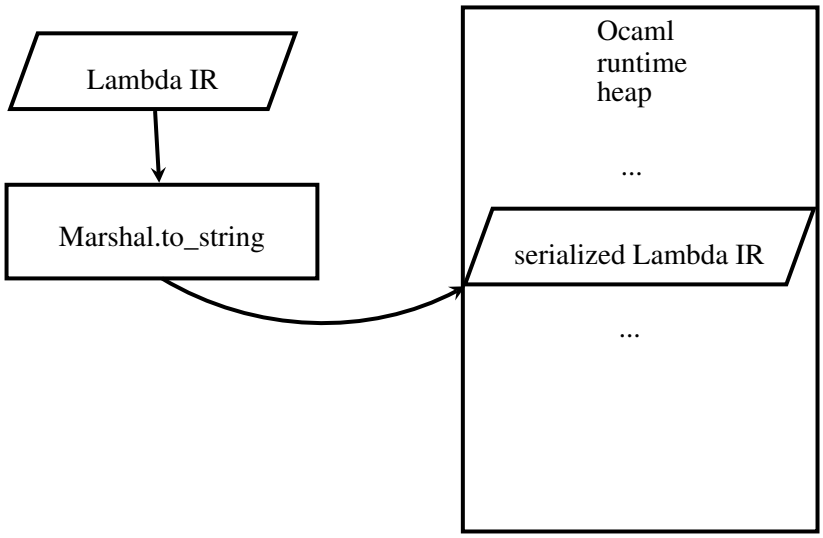

Figure 3.1: The implementation of code for closed terms

simply serialized (using the standard Marshal module), allocating a block of type string in the OCaml heap, as shown in Figure 3.1[1].

5. When run is applied to a value serialized by .<,>., the deserialized value becomes the body of a function that takes no arguments[2] in the Lambda intermediate representation. Conceptually, the run needs to call the load_lambda function in the toplevel (the OCaml toplevel is introduced in Section 2.5.2) passing it the zero-argument function to obtain a value. To allow run to actually call the load_lambda function, I added a C-level primitive to the OCaml run-time which calls a custom wrapper around the load_lambda function in the OCaml toplevel. The custom wrapper deserializes the serialized Lambda term, and passes the term to load_lambda. Figure 3.2 summarizes the operation of run.

[1] Serializing the code was an early design decision, intended to ensure that the entire structure of the Lambda terms is preserved

[2] Technically, OCaml does not support functions that take no arguments functions: a zero-argument function is represented in OCaml as a single-argument function whose argument is of special type unit. The unit type can be thought of as being similar to Java null

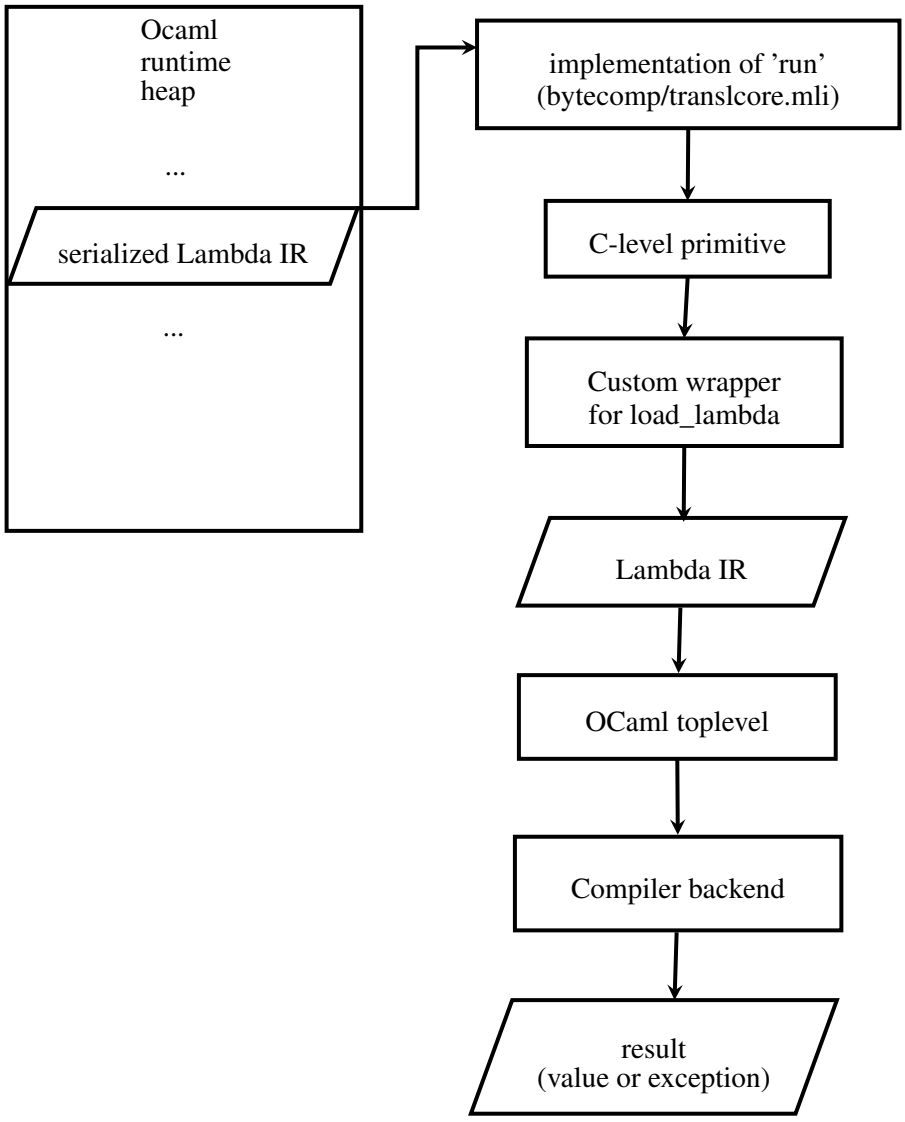

Figure 3.2: The implementation of run for closed terms

3.2 Cross-stage persistence

After settling on an implementation for run, the next item to tackle was *cross-stage persistence*. The need for cross-stage persistence can be illustrated with the simple function `let make_code a = .<a>.`. When the function is called, the desired behavior is that the piece of code returned from the function "remembers" the value of a that was in effect when make_code was called. From the standpoint of the user, the behavior is very similar to the behavior of make_adder in Section 2.5.1: when the function returned by make_adder is called, the function being called "remembers" the value of *n* that was passed to make_adder. Similarly, when `.!` is applied to the piece of code returned my make_code, the piece of code to which `.!` is being applied must "remember" the value of a that was in effect at the time make_code was called.

To support cross-stage persistence, it must be possible for run to execute a piece of code in a way that closely simulates how the code would have been executed if it had not been surrounded by the code brackets ($.<,>.$). Specificially, the code fragment inside the code brackets must have access to all the variables that are lexically visible at the point at which the code fragment occurs. In order to execute the code fragment in the same lexical environment in which the code fragment occurred, run needs to have access to a an environment that maps free variables in the code fragment to the values those variables were referring to when the expression that created the code fragment was encountered. One way to satisfy this requirement is to arrange things so that run has access to a closure created at the time the corresponding code construct is encountered. This is exactly the approach taken by the implementation presented here. Additionally, in order for run to be able to compile the code fragment, run needs to have access to a lexical environment that maps variable names to offsets in the closure block.

The responsibility for acquiring these two pieces of information is split between the implementation of code and implementation of run. As shown in Figure 3.3, the implementation of code is modified to preserve, in addition to the code fragment itself, the lexical environment in which the code fragment should be compiled.

The code construct allocates a structure which I will refer to as *MetaOCaml*

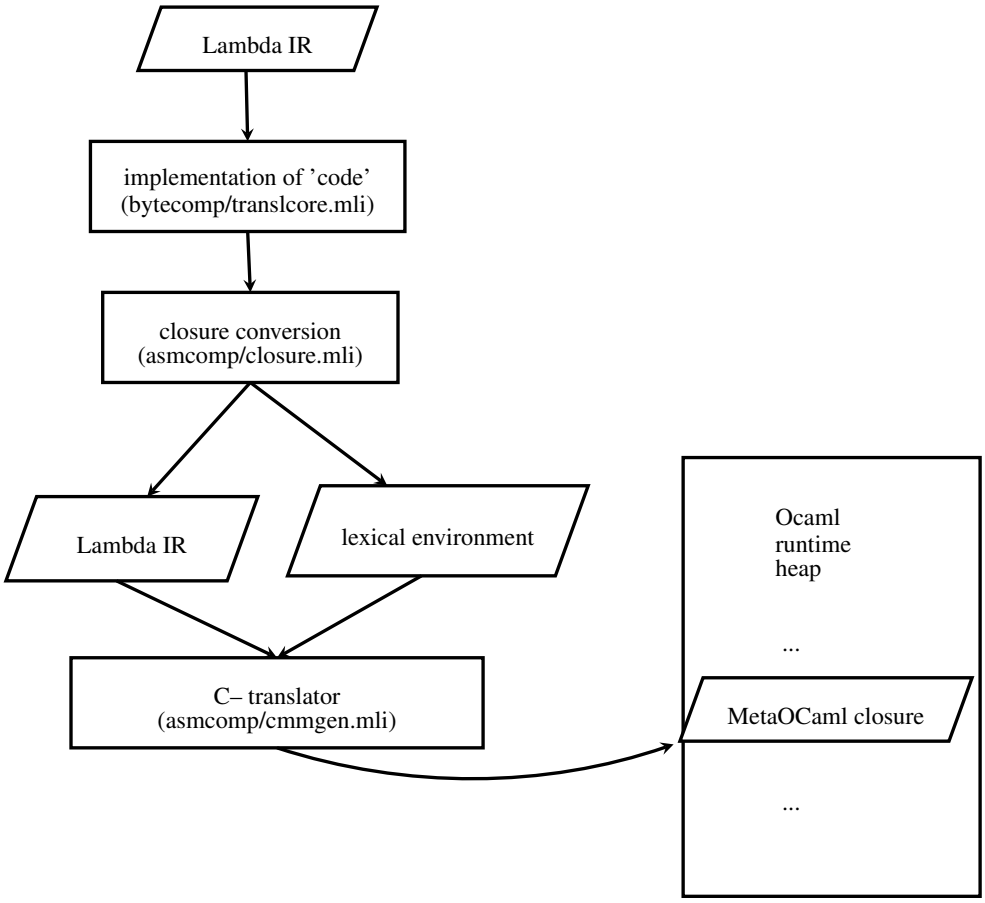

Figure 3.3: The implementation of code for cross-stage persistence

closure. A MetaOCaml closure is an extension of a regular OCaml closure[3]. The MetaOCaml closure, shown in Figure 3.4, adds two fields to the standard OCaml closure[4]:

1. The piece of code which contains the free variables whose values appear in the closure (as a Lambda intermediate representation).

2. The lexical environment that maps names of free variables to offsets in the

[3]Figure 2.14 shows the memory layout of a standard OCaml closure

[4]Conceptually, MetaOCaml closure adds two fields. In practice, the fields are stored as a serialized representation of a standard OCaml record, so the actual number of fields added is 1.

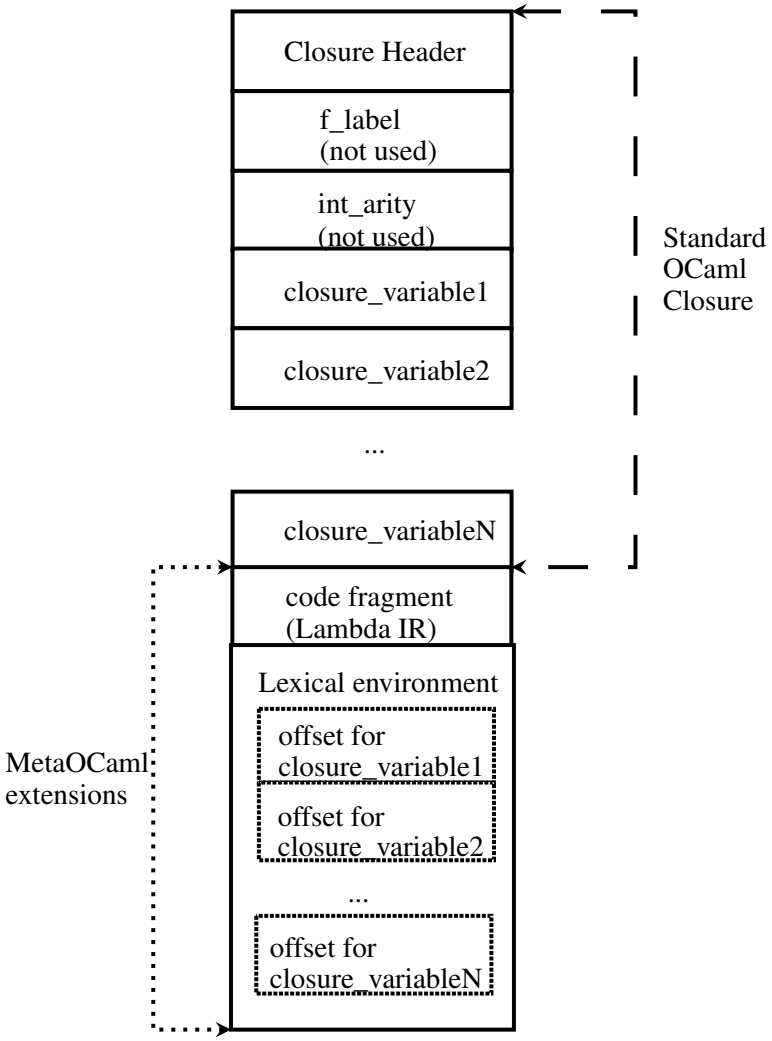

Figure 3.4: The MetaOCaml closure

closure.

As shown in Figure 3.5, run passes a pointer to a MetaOCaml closure to a C-level primitive, which, in turn, calls back into a custom wrapper around the `load_lambda` function in the OCaml toplevel. The custom wrapper uses the Lambda intermediate representation of the code fragment and the lexical environment it has retrieved from the MetaOCaml closure along with the pointer to

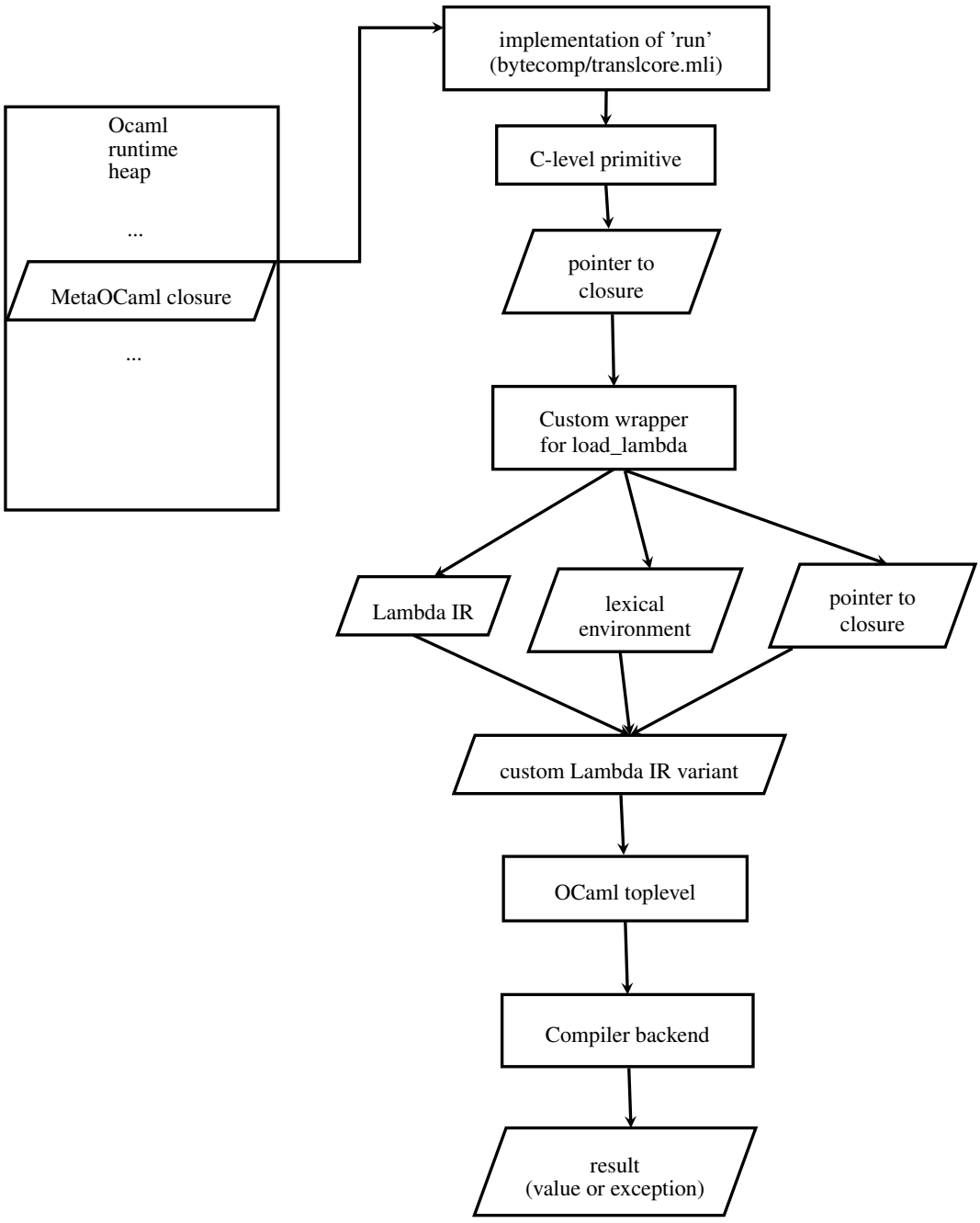

Figure 3.5: The implementation of run for cross-stage persistence

MetaOCaml closure to construct a custom variant record in the Lambda intermediate representation, which it passes to the `load_lambda` function in the OCaml toplevel. I extended the compiler backend to recognize the custom variant. The steps the compiler back-end takes to process the custom variant can be described as:

1. Perform closure conversion on the code fragment using the lexical environment it has received as part of the variant record. The result of the compilation is code for a function. The function takes zero parameters; it also takes a pointer to the environment in which it expects to be run.

2. Call the function obtained in the previous step passing to it the pointer to the closure contained in the variant record (and ultimately obtained from the C primitive) as the value of the environment parameter to obtain a result (i.e., a value or an exception).

3.3 Splicing

Splicing is the MetaOCaml mechanism by which pieces of code can be combined to produce other (larger) pieces of code. The basic idea is that the programmer constructs a template for the code she wants to have generated. The template (conceptually) has a few placeholders. The programmer then uses splicing to indicate what pieces of code to "plug into" the locations of the placeholders. The pieces of code being "plugged in" may have been passed as function's arguments or they may have been constructed by preceding code. It may help to think of MetaOCaml as embedding a language for manipulating pieces of code into OCaml. In the tradition of SICP [1], to understand the language one needs to understand what primitives it provides and what means for creating and constructing those primitives are available. The only primitive the language provides is code (a code fragment). The means for combining those primitives are quoting (putting the meta-brackets(. $<, >$.) around a literal code fragment) and splicing (plugging in a piece of code into a code fragment).[5] In Chapter 1, the splicing operator was

[5] I am treating run as somewhat tangential to the issue of constructing and manipulating pieces of code: run evaluates a piece of code to produce a value, but the resulting value may or may not be a piece of code.

used to combine the pieces of code that calculate the $n-1$st and $n-2$nd Fibonacci numbers. It is important to note that not every fragment of code preceded with the escape operator (`.~`) needs to be evaluated: Taha [30] introduces the notion of a *level* of a (syntactic) term, which is simply the number of meta-brackets the term is surrounded by minus the number of escape operators that precede the term. The rules of evaluation for MetaOCaml state that only escapes at level 0 need to be evaluated. To motivate this rule, it may help to think of evaluation as proceeding in phases, with subsequent phases being (potentially) separated in time. Only the splices that appear at level 0 are relevant for the evaluation during the current phase: the splices that appear at levels numbered higher than 0 are relevant for evaluation during later phases, which may potentially occur at later time.[6].

To summarize, the splicing operator may only occur inside the code brackets (`.<,>.`). The operator causes the expression that follows it to be evaluated. The result of the evaluation *must* be a piece of code[7]. That piece of code is "plugged in" at the location at which the splicing operator appears. The processing of splices only takes place for splices that appear at level 0: the splices that appear at levels higher than 0 will be evaluated during a later phase of evaluation.

The processing of a fragment of code with splices may be visualized as shown in Figure 3.6. The end goal of the processing is to produce a regular MetaOCaml closure, which can be processed in all the ways `code` can be processed. The processing proceeds as follows:

1. A code fragment is examined, and a list of splices it contains is created. References to splices inside the fragment of code are replaced with numbers representing position of respective splice in the list of splices.

2. The code for each splice is translated to the μLambda representation in the usual fashion. The code for the main body of the code is not translated at this time (though its lexical environment is preserved, to be used in the MetaOCaml closure allocated in the next step).

[6]The MetaOCaml type system prohibits splices at levels lower than 0

[7]The MetaOCaml type system ensures that the expression being spliced in is indeed a piece of code.

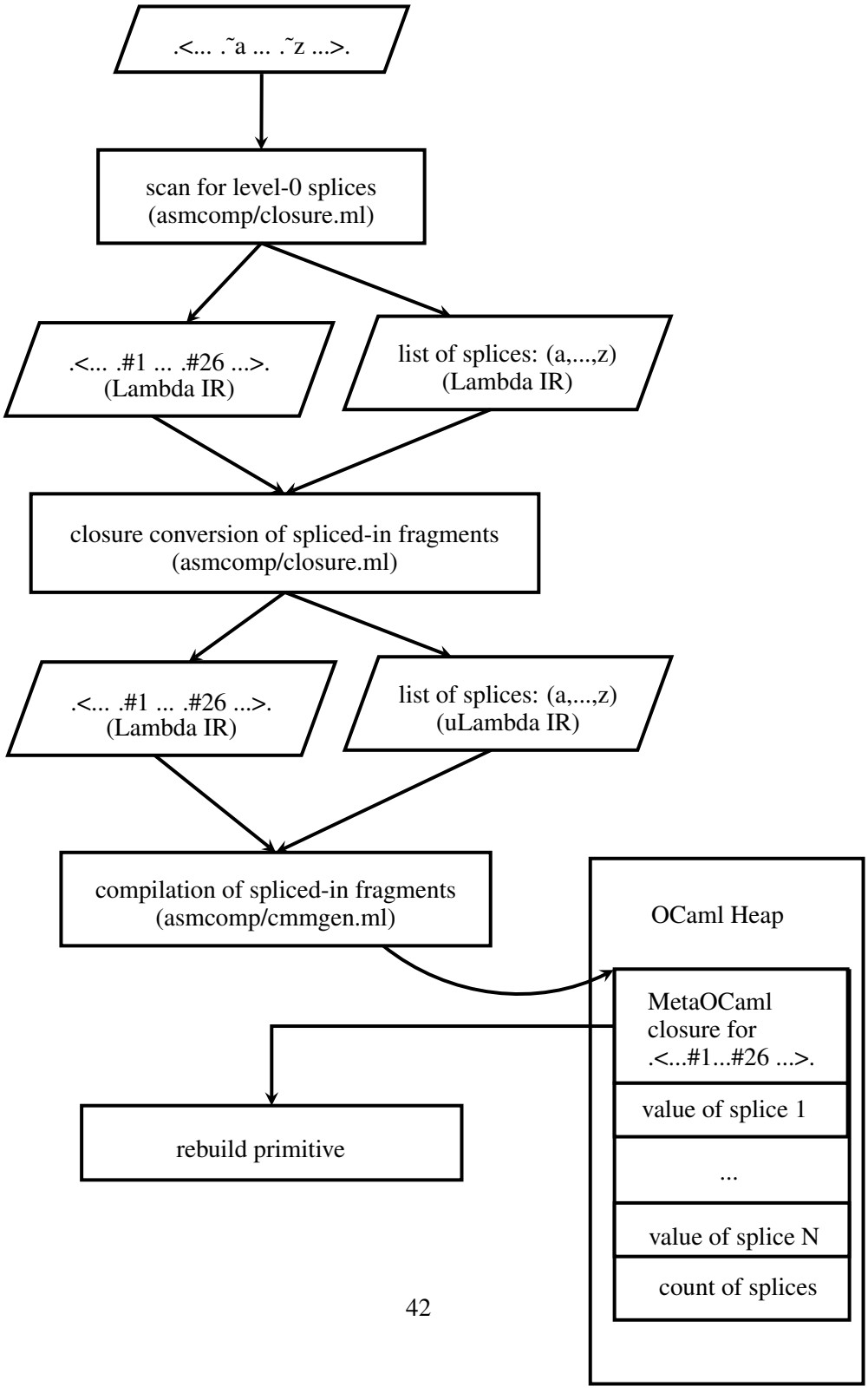

Figure 3.6: The implementation of code for splicing

3. A MetaOCaml closure is allocated for the main body of the code. The closure's size is declared to be large enough to hold the MetaOCaml closure proper, the values of splices, and a count of the number of splices.

4. The rebuild primitive is called on the extended MetaOCaml closure allocated in the previous step.

The data flow of the rebuild primitive is identical to the operation of the run primitive describe in Section 3.2: it constructs a custom variant of the Lambda intermediate representation and passes that variant to `load_lambda`[8], so I do not duplicate that part of the description here, and instead focus on how the custom variant of the Lambda intermediate representation constructed by the rebuild primitive is processed in the compiler backend. An important point to keep in mind is that the value of each splice is a MetaOCaml closure, which means that it contains the Lambda intermediate representation of a fragment of code, the vector of values of variables that are free in that code fragment, and a lexical environment mapping names of free variables to offsets in the values array.

The `rebuild` primitive operates on the extended MetaOCaml closure shown in Figure 3.6; it produces a regular MetaOCaml closure[9] as follows:

1. The lexical environment of the rebuilt MetaOCaml closure is the lexical environment of the MetaOCaml closure corresponding to the main code fragment augmented with names of variables that appear in the lexical environment of some splice but not in the lexical environment of the main code fragment.

2. The values vector of the rebuilt MetaOCaml closure is the values vector of the MetaOCaml closure corresponding to the main code fragment augmented with values corresponding to variable names that were added to the lexical environment of the MetaOCaml closure in the previous step.

3. The Lambda intermediate representation of the rebuilt closure is obtained by replacing the *i*th splice marker in the Lambda intermediate representation of

[8] It would have been possible to have a single primitive that dispatches on the type of value passed to it; having two primitives is a bit overkill, but it does offer a very straightforward implementation.

[9] Figure 3.4 illustrates a regular MetOCaml closure

```
let puzzle = .!(. <fun a ->
                  .~((fun x -> .<x>.) ((fun x -> .<a>.) 0)>.) 5
```

Figure 3.7: Example that illustrates the scope extrusion problem, by Taha

the main fragment with the Lambda intermediate representation found in *i*th splice[10]

As a result of this processing, a regular MetaOCaml closure is produced. This closure can then be processed in all the usual ways code can be processed, giving the user of the system multi-staging abilities.

3.3.1 Dealing with scope extrusion

The scope extrusion problem as it applies to MetaOCaml was described by Taha [30]. Briefly, the scope extrusion problem is the fact that a naive implementation of splicing may allow variables that should be bound (by an abstraction or let-form) to escape the scope of their corresponding binder. Taha's original example illustrating the scope extrusion problem appears in Figure 3.7. As first described in [30], a naive implementation of MetaOCaml will return something like `.<a123>.` when presented with this example. The peculiar value `.<a123>.` is the (uniquely-renamed) variable `a` which is bound by the `fun a ->`, but is free in the code fragment that follows the `.~`. The difficulty arises because, as mentioned in Section 3.3, the MetaML mental model of the order of evaluation demands that the spliced-in value be evaluated before the code fragment into which they are being spliced. The variable `a` is free in the code fragment, so the expression `(fun x -> .<a>.) 0` yields the variable `.<a123>.`; when the closure environment is built for the function `(fun x -> .<x>.)`, the variable `x` is bound to the code fragment that contains `.<a123>.`, and when the value bound to the variable `a` is finally known, a naive implementation does not "realize" that it needs to traverse the structure of the closure created for the function `fun a ->` to find and replace the now-unbound value of `a`.

My solution to the scope extrusion problem is based on the observation that, at any point in the code fragment, it is known which variables are bound (static scope

[10]The bound variables of each splice are renamed before substitution is performed to avoid accidental variable capture.

discipline ensures that that is the case). In other words, the compiler can find out which variables should be bound simply by recursively descending the code term while keeping track of the binders encountered so far and the variables they bind. To handle the scope extrusion problem in my code, I extended the Ulambda language by introducing a Ucover term (the inspiration for my implementation of Ucover as well as the term *cover* come from Taha [30]).

The high-level intuition for my implementation of the concept of covers, pioneered by Taha [30]) is that I want a construct that obeys the rules of regular lexical scoping (so that it can have access to all the variables that are lexically visible at the point at which it occurs), but which delays actually acting on the pieces of code it covers until the values of the variables that are appear to be free in the body of the code being spliced in are known.

The operation of the Ucover term can be described as follows:

1. When a piece of code is compiled, the Lambda-tree is traversed, looking for variables bound by functions. Each point in the code where an expression is spliced in is annotated with the list of (names of) variables that have been bound by the function definitions encountered so far. I view the information about "what variables are bound" as a property of the context into which an expression is being spliced rather than as an intrinsic property of the spliced-in expression itself.

2. When the expression being spliced is closure-converted, the lexical environment in which closure-conversion happens is "poisoned" by binding each of the variables that appear as an argument to any function (from the previous item) to a special tag (Ufreevar variable_name_here).

3. The closure-converted code is then examined to see which Ufreevars (if any) have been "closed over". For any Ufreevar's that have been captured, offsets from the beginning of the closure block are collected. (This trick is borrowed from stock OCaml close_functions code, which is what processes a "let rec" definition: that function does something very similar to what I described above to decide whether it actually needs to pass to a function the address of its corresponding closure block).

4. The list of Ufreevars is used to construct a Lcover that "wraps around" the splicing-in point. The Lcover expression is part of the "main" body of the code (as opposed to being part of the code of the expression being spliced in), so all variables that have been bound by functions are lexically visible to Lcover. Lcover eventually compiles to code that puts values corresponding to variables at their intended offsets in the closure's variable block (Ufreevar compiles to code that just allocates a dummy cell in the closure's variable block).

Chapter 4

Evaluation

In this chapter, I present the results of evaluating the performance of my MetaOCaml implementation and other MetaOCaml implementations on a few benchmarks. As mentioned in 2.4, a property of the MetaOCaml design is that it is always possible to erase all annotations from a piece of MetaOCaml code to end up with valid plain OCaml code. The role of the annotations is to allow the programmer to influence the order of evaluation. The stated motivation for design of MetaML [31] (and MetaOCaml) was that, by allowing programmers to control the order of evaluation, programmers will be able to influence the performance of the programs they write. In view of that design goal of annotations as a tool for improving performance of programs, I consider the ratio of running time for staged function (under MetaOCaml) to the running time of unstaged function (under the matching version of plain OCaml compiler) to be of primary interest, and raw running time statistics to be of secondary interest.

4.1 Methodology

The measurements for the 4.02.1 series of compilers were collected using the Core_bench library from Jane Street [17]. Suppose I have a *procedure* (i.e., a function that takes no arguments and does not return any interesting results to its caller) named p. The Core_bench library attempts to find and report the average running time t_{avg} of the procedure p which allows the total running time t_{total} for k runs of

```
open Core.Std
open Core_bench.Std

let p1 () = ...
let p2 () = ...

let main () =
  Command.run (Bench.make_command [
    Bench.Test.create ~name:"p1" p1;
    Bench.Test.create ~name:"p2" p2;
  ])

let () = main ()
```

Figure 4.1: The OCaml code to benchmark (hypothetical) procedures named p1 and p2 using Core_bench.

p to be calculated simply as $t_{total} = kt_{avg}$. It finds t_{avg} by sampling t_{total} for various values of k, and then fitting a straight line through the data points obtained during sampling. Figure 4.1 shows the OCaml code to benchmark two OCaml procedures named $p1$ and $p2$ using Core_bench.

Unfortunately, Core_bench is not available under the original MetaOcaml [4] (because the version of OCaml on which that implementation of MetaOCaml is based is too old), so I used a simple piece of code to run the function for a specific number of iterations and report the average running time. The code appears in Figure 4.2. The number of times to run each individual function was determined in an ad-hoc manner, by trying a few different values until the reported average time appeared to be stable; I report the number of iterations used for each function.

All measurements were performed on a virtual machine running Debian stable (jesse) release 8.2 with 64-bit Linux kernel version 3.16.0. The virtual machine was run under VirtualBox version 5.0.6; the virtual machine had a single CPU allocated to it; the host CPU was i5-2520M with the (nominal) clock frequency of 2.5 GHz.

Table 4.1 describes the microbenchmarks used to measure the performance of the OCaml systems under evaluation.

The power and fib are the traditional favorites of the partial evaluation and multi-stage programming communities, while the peval1 and peval2 were used by Taha [28] as case studies to motivate the usefulness of multi-stage programming techniques.

```
let iters = ref 1234

let rec loop i f =
  if i <= 0 then ()
  else begin f (); loop (i-1) f end

let simple_bench f =
  let t0 = Sys.time () in
  loop !iters f;
  let t1 = Sys.time () in
  print_float ((t1 -. t0) /. (float_of_int !iters)) ; print_newline ()

let main f =
  if Array.length Sys.argv > 1
  then iters := int_of_string(Sys.argv.(1));
  simple_bench f

let () = main (fun () -> ...)
```

Figure 4.2: The OCaml code to collect timing data from Taha *et. al* MetaOCaml.

Benchmark	Description
fib	The function to calculate Fibonacci numbers.
power	The function to raise a number to an (integer) power.
cfib	The staged version of the function to calculate Fibonacci numbers.
cpower	The staged version of the function to raise a number to an integer power.
peval1	An interpreter for a simple language with the four arithmetic operations and first-order functions.
peval2	A staged version of the interpreter for a simple language with the four arithmetic operations and first-order functions.

Table 4.1: Description of the benchmarks used in evaluating the performance of the MetaOCaml implementations.

The code for the power, fib, cpower and cfib appears in Figure 4.3. The code for peval1 and peval2 appears in [28][1].

4.2 Performance results

Table 4.2 shows the ratios of running times of staged code in MetaOCaml implementations being benchmarked to the respective running times of plain OCaml code under the same major and minor version of the compiler and the same backend as the staged code. My MetaOCaml implementation is currently based on OCaml 4.02.1 and comes with a native code backend, so I report the numbers normalized to native code produced by the standard OCaml compiler version 4.02.1; on the other hand BER MetaOCaml comes with a byte code backend (and is also based on OCaml 4.02.1), so I report numbers for BER normalized to byte code produced by the standard OCaml compiler version 4.02.1; finally the original MetaOCaml implementation is based on OCaml version 3.0.9, and comes with both a native code and a byte code backend – however, I am primarily interested in the native code backend, so I only report numbers for the original MetaOCaml normalized to the native code produced by the standard OCaml compiler version 3.0.9.

Tables 4.4, 4.5, 4.8, 4.6, 4.7, and 4.9 present the "raw" running times for BER MetaOCaml, my implementation of MetaOCaml, the original MetaOCaml, plain OCaml 4.02.1 (the byte code backend), plain OCaml 4.02.1 (the native code backend), and OCaml 3.09 (the native code backend) respectively. As mentioned in Section 4.1, the numbers for implementations based on OCaml version 4.02.1 were collected using Core_bench, while the numbers for the implementations based on OCaml 3.0.9 were collected using a simple piece of code which only measures the running times. Unfortunately, no numbers could be obtained for peval2 under the original MetaOCaml implementation, since the compiled program crashed with the message, "Fatal error: exception Ctype.Unify(_, _)."[2] (the program does run under the bytecode toplevel in the original MetaOCaml).

I speculate that the reason staged functions run significantly slower than un-

[1] For consistency, I prefix names of all staged functions with "c"(for "code"), so peval2 appears as cpeval2

[2] Regrettably, I am not familiar enough with the original MetaOCaml implementation to be able to effectives debug the cause of the crash.

Benchmark	BER MetaOCaml	My MetaOCaml	Original MetaOCaml
cpower	2850	$180 * 10^3$	6530
cfib	45000	$22 * 10^6$	$780 * 10^3$

Table 4.2: Ratio of running time of (annotated) benchmarks to the running time of unannotated program in the corresponding OCaml compiler.

```
let rec power x n = if n = 0 then 1 else x * power x (n-1)

let rec fib a b n = if n = 0 then a else fib b (a + b) (n-1)

let rec sfib a b n = if n = 0 then a else sfib b (.<(.~a) + (.~b)>.) (n-1)

let cfib = sfib (.<2>.) (.<3>.) 17

let rec spower x n = if n = 0 then .<1>. else .<(.~x) * (.~(spower x (n-1)))>.

let cpower = spower (.<2>.) 17
```

Figure 4.3: The code for power, fib and their staged counterparts, cpower and cfib

staged versions of the same function is that staged functions allocate significantly more heap memory than their non-staged counterparts. The memory allocation numbers, as reported by Core_bench are shown in Table 4.3. As can be seen, both BER MetaOCaml and my implementation of MetaOCaml allocate significantly more memory than regular OCaml when running these benchmarks.

I speculate that the reason my implementation allocates significantly more heap storage than the BER MetaOCaml is the serialization and deserialization of the `Lambda` intermediate language. Getting rid of the serialization and deserialization is one obvious avenue for future work. At a higher level, my measurements suggest that reducing the number of allocations in the run-time heap during a program's run time is likely to yield higher performance gains than reducing the number of CPU instructions the program executes while it runs.

Benchmark	BER MetaOCaml	My MetaOCaml	Stock OCaml
cpower	$18 * 10^3$	$2.5 * 10^6$	0
cfib	$2600 * 10^3$	$17.9 * 10^6$	0

Table 4.3: Memory allocation for benchmarks (in words allocated in the minor heap).

Name	Time	minor heap	major heap	minor $\rightarrow$ major promotions
cpower	1,600μs	$18 * 10^3$ words	$2 * 10^3$ words	$3 * 10^3$ words
cfib	230,000μs	$2,700 * 10^3$ words	$400 * 10^3$kw	$400 * 10^3$ words
cpeval2	600μs	$7 * 10^3$ words	$1 * 10^3$ words	$1 * 10^3$ words

Table 4.4: Raw data for BER MetaOCaml, as reported by Core_bench. All numbers are average values per run.

Name	Time	minor heap	major heap	minor $\rightarrow$ major promotions
cpower	110,000 μs	$2.5 * 10^6$ words	$5 * 10^3$ words	$3.7 * 10^3$ word
cfib	790,000 μs	$18 * 10^6$ words	$840 * 10^3$ words	$770 * 10^3$ words

Table 4.5: Raw data for my MetaOCaml implementation, as reported by Core_bench. All numbers are average values per run.

Name	Time	minor heap
power	575.46ns	
fib	510.08ns	
peval1	9,600ns	150.00w

Table 4.6: Raw data for OCaml v4.02.1 with byte code back-end, as reported by Core_bench. All numbers are average values per run. BER MetaOCaml running times are normalized to the numbers in this table.

Name	Time	minor heap
power	61ns	
fib	36ns	
peval1	1,800ns	149 words

Table 4.7: Raw data for OCaml v4.02.1 with native code back-end, as reported by Core_bench. All numbers are average values per run. The running times of my MetaOcaml implementation are normalized to the numbers in this table.

Name	Time	number of iterations
cpower	430μs	700
cfib	31,000μs	400
cpeval2	(crashed)	—

Table 4.8: Raw data for the original MetaOCaml implementation. All numbers are average values per run.

Name	Time	number of iterations
power	66ns	299999999
fib	41ns	199999999
peval1	1,690ns	29999999

Table 4.9: Raw data for OCaml 3.09 with native code backend. All numbers are average values per run. The running times of the originala MetaOcaml implementation are normalized to the numbers in this table.

Chapter 5

Conclusions

5.1 Summary

I have presented my re-implementation of MetaOCaml on top of a modern OCaml compiler. My implementation uses the Lambda intermediate representation (as opposed to abstract syntax trees) to represent OCaml code fragments. Cross-stage persistence is achieved by using an extended version of OCaml closures which combine the values vector of the usual OCaml closure with a code fragment to be compiled and the lexical in which environment to perform the compilation. Splicing is accomplished by directly manipulating the Lambda intermediate representation, followed by concatenation of the values vector of the OCaml closures which correspond to the main body of code and the code being spliced. (Implementing splicing by direct manipulation of Lambda terms is helped by the fact that Lambda language is a rather high-level and AST-like. It has been argued that this property makes certain kinds of "standard" optimizations awkward to perform [10], but the AST-like nature of Lambda definitely helped with my implementation of MetaOCaml). The scope extrusion problem is dealt with by a pre-emptive scan of the lexical structure of each code term.

5.2 Retrospective

I have presented a (re)-implementation of MetaOCaml that supports turn-key native code generation on top of a modern OCaml compiler infrastructure. The existing MetaOCaml implementation that supports turn-key generation of native code are not based on a modern OCaml compiler. The existing MetaOCaml implementation that is based on a modern OCaml compiler does not support turn-key generation of native code.

Along with producing a convenient way to compile MetaOCaml to native code, a secondary goal of the project was to evaluate the current version of OCaml as a substrate for implementing MetaOCaml. In summary, I am happy to report that the current OCaml compiler is a very good environment for implementing MetaOCaml: the total number of lines of code that needed to be changed or added as compared with the standard OCaml compiler is well under 3000. I was able to extend the existing compiler data structures and functions essentially without doing any major re-architecturing work on the compiler: outside of exposing the load_lambda interface in `toploop/nattoploop.ml`, and changing the scope of the module-global reference program_name in the same file, very little of my implementation work can be accurately described as "fixing bugs in the OCaml compiler."

A third goal of the project was to check whether by removing the redundant type-checking at run-time the overall performance of the MetaOCaml compiler was improved (as compared with the existing native-code compiler). Unfortunately, no significant improvements in run time of programs were observed during the performance evaluation. One area potentially worth exploring with respect to run-time performance is whether serializing and deserializing of complete Lambda strcutures is really necessary. It certainly was a convenient and quick implementation strategy, but it may have come at a cost in terms of program performance.

5.3 Possible future work

Possible directions for future work are:

Do not serialize `Lambda` A question worth investigating is whether performance can be significantly increased by getting rid of serialization/deserialization of the Lambda intermediate representation into string.

Bytecode compiler During the development of the native code compiler, I implemented `run` and cross-stage persistence in a byte code compiler. Unfortunately, the implementation of the byte code version of those features did not use the implementation model which can be summarized as "extend the Lambda language enough to be able to support MetaOCaml; make a minimal wrapper around `load_lambda`": instead, it relied on a special version of `load_lambda`. Redoing that implementation so that it does not need a special version of `load_lambda` would be desirable.

Native code back-end for BER MetaOCaml BER MetaOCaml is an implementation of MetaOCaml that keeps pace with current developments of the OCaml compiler. Some of the experience gained from the current "clean-room" implementation of native MetaOCaml code generator may be transferrable towards producing a native code generator for BER.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. 2nd. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0262011530.

[2] Baris Aktemur et al. "Shonan Challenge for Generative Programming: Short Position Paper". In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*. PEPM '13. Rome, Italy: ACM, 2013, pp. 147–154. ISBN: 978-1-4503-1842-6.

[3] Andrew W. Appel. *Compiling with Continuations*. New York, NY, USA: Cambridge University Press, 1992. ISBN: 0-521-41695-7.

[4] Cristiano Calcagno et al. "Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection". In: *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*. GPCE '03. Erfurt, Germany: Springer-Verlag New York, Inc., 2003, pp. 57–76. ISBN: 3-540-20102-5. URL: http://dl.acm.org/citation.cfm?id=954186.954190.

[5] Luca Cardelli. *The Functional Abstract Machine*. Technical Report TR-107. Bell Laboratories, 1983. URL: https://karczmarczuk.users.greyc.fr/matrs/Maitrise/fam.pdf.

[6] A. Church. *The calculi of lambda-conversion*. Annals of mathematics studies. Princeton University Press, 1941.

[7] Joshua Dunfield. *private communication*. 2015.

[8] Matthias Felleisen and Daniel P. Friedman. *A Little Java, a Few Patterns*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1998. ISBN: 0-262-56115-8.

[9] Matthias Felleisen et al. *How to Design Programs: An Introduction to Programming and Computing*. Cambridge, MA, USA: MIT Press, 2001. ISBN: 0-262-06218-6.

[10] Fabrice Le Fessant. *Flambda trunk*. Dec. 18, 2014. URL: https://github.com/ocaml/ocaml/pull/132 (visited on 12/18/2014).

[11] Matthew Flatt. *Rebuildxing Racket's Graphics Layer*. June 15, 2015. URL: http://blog.racket-lang.org/2010/12/racket-version-5.html (visited on 12/08/2010).

[12] Free Software Foundation, ed. *GNU Emacs Lisp Reference Manual*. section GNU Emacs Internals, subsection Object Internals. 2015. URL: http://www.gnu.org/software/emacs/manual/html_node/elisp/Object-Internals.html#Object-Internals.

[13] Free Software Foundation, ed. *The Guile Reference Manual*. section Guile Implementation. 2014. URL: http://www.gnu.org/software/guile/manual/html_node/Data-Representation.html#Data-Representation.

[14] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 0130305529.

[15] Paul Graham. *Succintness Is Power*. Sept. 15, 2015. URL: http://www.paulgraham.com/power.html (visited on 05/01/2002).

[16] Miguel Guerrero et al. "Implementing DSLs in metaOCaml". In: *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '04. Vancouver, BC, CANADA: ACM, 2004, pp. 41–42. ISBN: 1-58113-833-4.

[17] Roshan James. *Core_bench: better micro-benchmarks through linear regression*. May 17, 2014. URL: https://blogs.janestreet.com/core_bench-micro-benchmarking-for-ocaml/ (visited on 10/14/2015).

[18] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 0-13-020249-5.

[19] Oleg Kiselyov and Chung-chieh Shan. "The MetaOCaml files". 2010.

[20] Oleg Kiselyov and Walid Taha. "Relating FFTW and Split-radix". In: *Proceedings of the First International Conference on Embedded Software and Systems*. ICESS'04. Hangzhou, China: Springer-Verlag, 2005, pp. 488–493. ISBN: 3-540-28128-2, 978-3-540-28128-3.

[21] Peter J Landin. "The mechanical evaluation of expressions". In: *The Computer Journal* 6.4 (1964), pp. 308–320.

[22] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990.

[23] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782.

[24] Simon Peyton Jones and Norman Ramsey. *C-- FAQ (Web Archive)*. 2007. URL: http://web.archive.org/web/20080716105923/http://www.cminusminus.org/faq.html.

[25] Simon Peyton Jones and Norman Ramsey. *C-- Official Site (Web Archive)*. 2007. URL: http://web.archive.org/web/20080822062234/http://www.cminusminus.org/.

[26] Eric S. Roberts. *Thinking Recursively: With Examples in Java*. John Wiley & Sons, 2005. ISBN: 0471701467.

[27] Moses Schönfinkel. "Über die Bausteine der mathematischen Logik". In: *Math. Ann* 92.3–4 (1924), pp. 305–316.

[28] Walid Taha. "A gentle introduction to multi-stage programming". In: *Domain-specific Program Generation, LNCS*. Springer-Verlag, 2004, pp. 30–50.

[29] Walid Taha. "A Sound Reduction Semantics for Untyped CBN Mutli-stage Computation. Or, the Theory of MetaML is Non-trival (Extended Abstract)". In: *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '00. Boston, Massachusetts, USA: ACM, 1999, pp. 34–43. ISBN: 1-58113-201-8.

[30] Walid Mohamed Taha. "Multistage Programming: Its Theory and Applications". AAI9949870. PhD thesis. 1999. ISBN: 0-599-52497-9.

[31] Walid Taha and Tim Sheard. "Multi-stage Programming with Explicit Annotations". In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '97. Amsterdam, The Netherlands: ACM, 1997, pp. 203–217. ISBN: 0-89791-917-3.

[32] Mads Tofte et al. *Programming with Regions in the MLKit (Revised for Version 4.3.0)*. Tech. rep. IT University of Copenhagen, Denmark, Jan. 2006.

[33] Mitchell Wand. "The Theory of Fexprs is Trivial". In: *Lisp Symb. Comput.* 10.3 (May 1998), pp. 189–199. ISSN: 0892-4635.