清 华 大 学

# 综 合 论 文 训 练

题目：类 Caml 语言函数式编译器的实现


系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：宋方睿

指导教师：陈文光教授


2015 年 6 月 29 日

# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

**(涉密的学位论文在解密后应遵守此规定)**

签　名：_____ 导师签名：_____ 日　期：_____

# 中文摘要

Caml 是通用函数式编程语言 ML 家族的一个方言，本项目实现了一个 Caml 编译器 Caml Featherweight，实现了除模块系统外几乎全部的的 Caml 语法。项目主体采用 OCaml 编写，生成抽象机器字节码后解释执行，运行时系统和字节码解释器使用 C。项目还实现了一个代码排版工具，并把 Caml Featherweight 移植到了 JavaScript，可以在网页浏览器里编译执行。

**关键词**：编译器；函数式编程语言；Caml；ML

# ABSTRACT

Caml is a variant of general purpose functional programming language ML. This project implements a Caml compiler: Caml Featherweight, which implements almost all Caml syntax. The main part of the compiler is written in OCaml, generates bytecode on an abstract machine to execute. The runtime system and bytecode interpreter is written in C. A code pretty printer is also implemented. Caml Featherweight has also been ported to JavaScript which can be executed in web browsers.

**Keywords:** compiler; functional programming language; Caml; ML

# 目 录

# 第 1 章　引言

Caml 是 ML 编程语言家族的一门方言，由 INRIA[①] 开发，之前在 École Normale Supérieure。Caml 是个一门通用的函数式编程语言，有如下几个特点：

- 采用 call-by-value 的求值策略，允许副作用和 imperative programming。
- 使用垃圾收集进行自动内存管理。
- 支持 first-class function，函数可以像 `int`、`string` 那样的值一样自由传递、创建。
- 实现了静态类型检查，并能自动推导类型，无需 C 中的类型注解。
- 提供了 immutable programming 的良好支持。
- 参数多态，提供了一些通用编程支持，类似于 Java 中的 generic。
- 支持代数数据类型和模式匹配以处理复杂的数据结构。

本项目是[1] 提到的 Zinc 的实验的一个实现，实现了除模块系统外几乎全部的的 Caml 语法，在一些细节上有改动，因此称之为类 Caml 语言，库函数支持比较少。

Caml 的最初实现于 1987 年出现，使用了 Lisp，因为内存和 CPU 需求大而被戏称为 “Heavy CAML”。在 1990 和 1991 年，Xavier Leroy 编写了新的实现 Caml Light，运行时系统和字节码解释器使用 C 编写，其余则是 Caml。1995 年，Caml Special Light 发布，它在很多方面扩展了 Caml Light，这一项目之后吸收了很多面向对象的概念并被重命名为 Objective Caml(之后改名为 OCaml)。

本项目用 OCaml 编写，参考了 Caml Light 的结构，运行时系统和字节码解释器使用 C 编写，有 1000 多行，其他部分都是 OCaml，有 5000 多行，规模比 Caml Light 小很多，因此取名为 Caml Featherweight。

① French Institute for Research in Computer Science and Automation

# 第 2 章　项目实现的类 Caml 语言语法

项目实现的类 Caml 语言和 Caml 基本完全一致，因此下面就用 Caml 来指代本项目编译器实现的语言。

## 2.1　Caml

Caml 程序由若干定义和表达式组成，顶层结构有四种形式：

- 自定义类型声明。用于创建新的代数数据类型，或者之前已定义类型的别名。
- let 定义。类似于 C 中定义全局变量或函数。函数式编程语言中函数是 first-class 的，和值没有太多区分，因此值和函数的定义采用了相似的形式。
- 自定义异常类型。
- 表达式。

Caml 程序可以看作是依次解析并执行各个顶层结构，因此遇到表达式时会立即求值。在 Caml 中没有表达式和语句的区分，表达式可以产生各种各样的副作用。

## 2.2　概览

下面来看一个例子 (里面用到的一些函数 Caml Featherweight 并没有实现，只能解析这段代码)，Trabb Pardo-Knuth algorithm 的 Caml 实现如下：

```
let f x = sqrt x +. 5.0 *. (x ** 3.0)
let p x = x < 400.0

let () =
  print_endline "Please enter 11 Numbers:";
  let lst = Array.to_list (Array.init 11 (fun _ ->
    read_float ())) in
  iter (fun x ->
    let res = f x in
    if p res
    then printf "f(%g) = %g\n%!" x res
```

```
    else eprintf "f(%g) :: Overflow\n%!" x
  ) (rev lst)
```

前两行 `let` 是定义了两个函数，分别名为 `f` 和 `p`，都接受一个名为 `x` 的参数，函数体在等号右边。Caml 使用空格表示函数应用，因此 `sqrt x` 就类似于 C 中的 `sqrt(x)`，并且函数应用的拥有很高的结合性，因此 `sqrt x` 的计算发生在加法之前。`+.` 是浮点数加法，`**` 是乘方。

`let () =` 后是一个表达式，该表达式的最外层操作符是`;`，C 当中用于分割语句的`;` 在 Caml 里是表达式的组成单元，效果是对先对前一个表达式求值，再对后面的表达式求值，达到顺序求值的效果。从中可以体会到 ML 设计的优雅之处，把语句和表达式统一起来。

`;` 前的 `print_endline` 用于产生输出，后面则是一个 `let` 绑定，类似于 C 中在一对大括号里定义局部变量并执行后续语句。`iter` 接受两个参数，第一个参数是一个 lambda 函数 (匿名函数)，该参数自身也是一个接受一个参数的函数。从这里可以看到 Caml 中函数作为 first-class 对象的特征：可以被方便的创建并传递。

## 2.3 基本类型

Caml 支持五个基本类型，各类型都有字面值的表示方法：

表 2.1 基本类型

| | |
|---|---|
| `bool` | `true`、`false` |
| `char` | `'a'`、`'\n'` |
| `int` | `12`、`0x0e` |
| `float` | `3.0`、`2e-5` |
| `string` | `"hello"`、`"world"` |

## 2.4 操作符

表2.2列出了 Caml 支持的操作符：

其中 `=`、`<>`、`<` 等操作符，使用了值的运行时表示来判断两个任意类型参数的大小，实现可以参考 `runtime/compare.c`。

表 2.2　操作符

| | | |
|---|---|---|
| = | 'a -> 'a -> bool | 相等，支持任意类型参数的比较 |
| <> | 'a -> 'a -> bool | 不等 |
| == | 'a -> 'a -> bool | 对于 int：是否相等；其他，是否同一对象 |
| != | 'a -> 'a -> bool | != 结果取反 |
| <, | 'a -> 'a -> bool | 小于，支持任意类型参数的比较 |
| <= | 'a -> 'a -> bool | 小于等于 |
| >, | 'a -> 'a -> bool | 大于 |
| >= | 'a -> 'a -> bool | 大于等于 |
| + | int -> int -> int | 加 |
| - | int -> int -> int | 减 |
| * | int -> int -> int | 乘 |
| / | int -> int -> int | 除 |
| mod | int -> int -> int | 模 |
| +. | float -> float -> float | 加 |
| -. | float -> float -> float | 减 |
| *. | float -> float -> float | 乘 |
| /. | float -> float -> float | 除 |
| && | bool -> bool -> bool | 逻辑与 |
\| \| | bool -> bool -> bool | 逻辑或 |
| land | int -> int -> int | 按位与 |
| lor | int -> int -> int | 按位或 |
| lxor | int -> int -> int | 按位异或 |
| lsl | int -> int -> int | 逻辑左移 |
| lsr | int -> int -> int | 逻辑右移 |
| asr | int -> int -> int | 算术右移 |
| not | bool -> bool | 逻辑否 |
| + | int -> int | 单目 |
| - | int -> int | 单目，相反数 |
| +. | float -> float | 单目 |
| -. | float -> float | 单目，相反数 |

float 的加减乘除操作符和 int 不同[2]，这里也可以看到 ML 家族 Caml 和 SML 两大流派的分歧。SML 采用同一套操作符，这样就需要操作符重载；而 Caml 采纳了 “principled overloading or none at all” 的哲学。

## 2.5 元组

元组是几种类型的复合，可以看成几种不同类型的有序集合。元组的各个成分用逗号分割，外面套圆括号，比如：

```
# let a_tuple = (3,"three");;
val a_tuple : int * string
```

某些不引起歧义的地方可以省略圆括号，规则比较琐碎，这里不表。

可以用模式匹配抽取元组的各个成分：

```
# let (x,y) = a_tuple;;
val x : int
val y : string
```

## 2.6 列表

元组可以把固定数目的不同元组聚合在一起，列表则可以表示相同类型的任意数目元素，语言提供的表示列表字面量的语法，如表示空列表的 `[]`，带有两个元素的列表 `[3; 4]`。

列表可以看成是一个单链表，单个节点可以为空 (`[]`)，或者包含一个元素和一个指向后继节点 (对应的构造器为`::`，比如：`3::[]`)。`::` 构造器是一个操作符，是右结合的，之前提到的 `[3; 4]` 可以看作 `3::4::[]` 的语法糖。

## 2.7 数组

数组类似于列表，可以表示相同类型的任意数目元素，并提供了随机访问的功能：

```
let a = [|3; 4; 7|]
let b = [|"a",('c',false); "b",('d',true)|]
```

访问数组中的一个元素：

```
a.(3)
```

修改数组中的一个元素：

```
a.(3) <- 'a'
```

## 2.8 函数

接受一个参数，返回参数加一的函数如下：

```
let plus_one x = x + 1
```

和类 C 语言不同，函数名和参数间用空格分割，并且函数体前使用 =，因为 ML 中没有像 C 那样区分语句和表达式。

如果要定义多参数的函数，参数间也用空格分割：

```
let sum_of_three x y z = x + y + z
```

函数允许自递归，这时需要使用 let rec：

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

如果要定义互递归的函数，必须在一个 let rec 中同时定义各个函数，它们之间用 and 分割：

```
let rec is_even x =
  if x = 0 then true else is_odd (x-1)
and is_odd x =
  if x = 0 then false else is_even (x-1)
```

这是 Caml 语言设计的一种简化。C 中可以用函数声明的方式使得定义在前面的函数体内可以引用之后定义的函数，但需要对声明和定义做相容性检查，否则可能在运行时产生类型不一致的问题。在翻译代码时，Caml 由于使用了递归函数必须同时定义的限定，就只需对顶层结构做一次处理。

## 2.9 代数数据类型和模式匹配

Caml 内置了一些类型构造器。option 代表可能缺失的值，如 None、Some (4,true,7,'a')。

下面例子是 option 的一个应用，表示带外值：

```
let divide x y =
  if y = 0 then None
  else Some(x/y)
```

顶层结构的 type 可以用于自定义代数数据类型：

```
type t = Zero | One of int | Two of char * int
type tt = A of t
```

上面定义了一个类型 t，它有三个构造器：Zero、One 和 Two，接受不同的参数。

模式匹配和代数数据类型配套使用，可以抽取构造器的参数：

```
match a with
| A t ->
    match t with
    | Zero -> 0
    | One i -> i
    | Two(ch,i) -> i
```

模式匹配也可用于 int、string 等基本类型。_ 是通配符，可以匹配任意值：

```
match s with
| "hello" -> true
| _ -> false
```

模式匹配也可用于复杂的类型，各分支中以以小写字母开头的表示符表示变量，可以匹配任意值，从而可以在 -> 后的表达式中引用：

```
match s, 'a', 3+2 with
| "a", 'a', v -> "a"
| g, h, 5 -> g
| _ -> "b"
```

## 2.10 异常

异常提供了一种中断当前的计算，报告错误，捕获错误的机制。

除数为零时会触发内置的 Division_by_zero 异常，下面的程序捕获了该异常并输出 12：

```
try
  output_int (1/0)
with Division_by_zero ->
  output_int 12
```

如果模式匹配失败，则会产生 Match_failure 异常：

```
output_int (try
  let Some 3 = None in
  1
with Match_failure _ ->
  0)
```

try 实质上形成了一个动态作用域，如果最近的 try 无法捕获的话，异常会交给外层的 try：

```
output_int (try
  try
    let Some 3 = None in
    1
  with Division_by_zero ->
    0
with Match_failure _ ->
  2)
```

# 第 3 章　工具链使用说明

cd src/后执行 make 生成字节码编译器 camlfwc、字节码解释器 camlfwrun、字节码查看器 camlfwod。

编译并链接源文件得到可执行的字节码文件：

```
% ./camlfwc -v /tmp/a.ml -o /tmp/a.out
Compiling /tmp/a.ml
- : unit
Linking /tmp/a.zo -> /tmp/a.out
```

使用 camlfwrun 执行：

```
% ./camlfwrun /tmp/a.out
```

camlfwod 可以显示目标文件或字节码可执行文件的指令列表。

```
% ./camlfwod /tmp/a.zo
%File /tmp/a.zo
Relocatable file

Offset 8
Length 33
0008: PUSHMARK
...

% ./camlfwod /tmp/a.out
File /tmp/a.out
Executable

Length 34
000c: PUSHMARK
%File /tmp/a.zo
```

```
Relocatable file

Offset 8
Length 33
0008: PUSHMARK
...
```

# 第 4 章　编译器的设计

## 4.1　执行方法

编译器的执行方法大致分成四类：

1. 解释执行语法树。容易实现，解析得到语法树后再做较少的工作即可实现。运行时解释执行，但性能很低。
2. 生成本地代码，处理器可以直接执行。优点是性能较高。
3. 生成在抽象机器上执行的字节码。类似于本地代码，但生成的代码基于一个抽象机器。优点是移植性强，只需把运行时系统移植到目标架构上即可。
4. 翻译到其他高级语言。和生成抽象机器的字节码类似，但前者过于低级，实现很多结构需要大量指令。编译到一门高级语言在很多地方上能得到简化。

## 4.2　语法树设计和类型信息表示

### 4.2.1　表达式

我需要一个表示表达式的类型，为了提供更具体的解析错误信息，需要把位置信息放在附在节点信息里。表示位置信息有两种方法，一种是在所有构造器里设置一个域表示：

```
type expression =
  | Pexpr_apply of location * expression * expression list
  | Pexpr_array of location * expression list
  | ...
```

这样在所有模式匹配的地方都要匹配位置域，很不方便。如果把构造器都改成 record 类型的话，模式匹配也会不方便。

比较好的方式是拆分为两个类型：

```
type expression = { e_desc: expression_desc; e_loc: location }
and expression_desc =
  | Pexpr_apply of expression * expression list
  | Pexpr_array of expression list
```

如果前端工作复杂的话，那么 expression 还可以考虑设计成参数化的。但在我的实现中只需要使用到一种语法树表示，没必要使用参数化。

```
type expression = { e_desc: expression_desc; e_loc: location }
and expression_desc =
  | Pexpr_apply of expression * expression list
  | Pexpr_array of expression list
  | Pexpr_constant of constant
  | Pexpr_constraint of expression * type_expression
  | Pexpr_constr of long_ident * expression option
  | Pexpr_for of string * expression * expression * bool * expression
  | Pexpr_function of (pattern * expression) list
  | Pexpr_ident of long_ident
  | Pexpr_if of expression * expression * expression option
  | Pexpr_let of bool * (pattern * expression) list * expression
  | Pexpr_sequence of expression * expression
  | Pexpr_try of expression * (pattern * expression) list
  | Pexpr_tuple of expression list
```

### 4.2.2 模式匹配

模式匹配和表达式共享了一些语法单元，比如字面量、元组、构造器、变量等，但其中不允许出现 for、; 等，和表达式还是有一些差别，因此需要为它定义一个类型 pattern。

pattern 是个和 expression 平级的概念，同样出于错误信息的考量，添加位置信息并拆分为两个类型：

```
and pattern = { p_desc: pattern_desc; p_loc: location }
and pattern_desc =
  | Ppat_alias of pattern * string
  | Ppat_any
  | Ppat_array of pattern list
  | Ppat_constant of constant
  | Ppat_constraint of pattern * type_expression
  | Ppat_constr of long_ident * pattern option
  | Ppat_or of pattern * pattern
  | Ppat_tuple of pattern list
  | Ppat_var of string
```

### 4.2.3 类型表达式

expression 和 pattern 中允许出现类型限定 (如 (3 : int)、(xs : 'a list))，用 type_expression 表示，这也是一个和 expression 平级的概念。

```
and type_expression = { te_desc: type_expression_desc;
  te_loc: location }
and type_expression_desc =
  | Ptype_var of string
  | Ptype_arrow of type_expression * type_expression
  | Ptype_tuple of type_expression list
  | Ptype_constr of long_ident * type_expression list
```

#### 4.2.4 Type constructor、具体类型

解析阶段以外用到的、类型相关。由于 Caml 支持参数化类型，因此如何表示类型、构造器等的描述变得很复杂。

考虑如何统一表示无参数的类型 (如 `int`、`string`) 和参数化类型 (如`'a option`)，它们都是由 type constructor(`int`、`string`、`option`) 接受零或多个参数得到的，type constructor 可以看作是一个独一无二的标识符，我用 `type_constr` 表示。

具体的类型用 `typ` 表示，可以由 type constructor 应用到参数上得到 (`Tconstr`)，也可以通过 product 类型或 arrow 类型等方式得到，也可以是一个 generic 类型 (`Tvar`，`typ_level` 为 -1) 或者是一个尚未确定的类型 (`Tvar`，`level` 非 -1)。

另外 Caml 支持 type abbreviation，type constructor 可能指向另一个具体类型。

综合上述考量得到下面的定义：

```
type typ = { typ_desc: typ_desc; mutable typ_level: int }
and typ_desc =
  | Tarrow of typ * typ
  | Tconstr of type_constr global * typ list
  | Tproduct of typ list
  | Tvar of typ_link ref
and typ_link =
  | Tnolink
  | Tlink of typ
and type_constr = { ty_stamp: int; mutable ty_abbr: type_abbrev }
and type_abbrev =
  | Tnotabbrev
  | Tabbrev of typ list * typ
```

#### 4.2.5 类型描述和构造器描述

对于类似 `type t = A` 这样的类型定义，以及一些内置类型 (如 `float`) 等，用 `type_desc` 表示，并存储在一个全局的表里，供编译的后续阶段引

用。参数化类型允许参数为任意类型，因此 `type_desc` 中只需要记录参数个数 (`ty_arity`)。等号右边的描述则用 `type_components` 类型表示，可以是 abstract type、variant type 或是 type abbreviation。

与 `type_desc` 关联的是构造器描述 `constr_desc`，表达式或模式匹配中出现构造器时会用到这个类型，它需要的域是所表示的具体类型 (`typ`)，是否接受参数 (`cs_kind: constr_kind`) 及接受的参数的类型 (`of` 后面的) 等。另外需要一个域 `cs_tag: constr_tag` 用于区分同一个具体类型的不同构造器。

```
type type_desc =
  { ty_constr: type_constr global; ty_arity: int;
    mutable ty_desc: type_components }
and type_components =
  | Abstract_type
  | Variant_type of constr_desc global list
  | Abbrev_type of typ list * typ

and constr_desc =
  { cs_arg: typ; cs_res: typ; cs_tag: constr_tag;
    cs_kind: constr_kind }

and constr_kind =
  | Constr_constant
  | Constr_regular
  | Constr_superfluous of int
```

其中 `constr_tag` 描述构造器的类型。考虑模式匹配的代码生成场景，我需要一种办法能区分同一个类型的不同构造器。如果构造器是 closed type 的，那么区分不同的构造器只需要一个信息，即构造器的标号，同一个类型的不同构造器标号互不相同。为了代码生成方便，构造器的数目也编码到标号中，可以用一个元组 `int * int` 来表示标号。

为了一致性，`exception` 类型也应看作构造器，而 `exception` 类型是开放的，即构造器可以定义在不同地方，数目不受限制，可以实现为一种 extensible type，这些类型用标识符表示。

```
type constr_tag =
  | Constr_tag_regular of int * int
  | Constr_tag_extensible of long_ident * int
```

## 4.3 词法分析

### 4.3.1 词法分析

使用工具 ocamllex。这一部分实现在 `lexer.mll` 中。

Caml 支持自定义操作符，它们的优先级由第一个字符决定，优先级会对语法分析产生影响，因此把优先级不同的操作符表示为不同的词素。

Caml Light 没有对大写和小写开头的标识符进行区分，但参阅 OCaml 源码可以看到区分后语法分析会更加容易。因此我的项目中也按大小写进行区分。

还有一些需要注意的地方是字符串、字符、注释等的解析，需要考虑转义字符等问题，但都不是难点。

### 4.3.2 语法分析

这一部分实现在 `parser.mly` 中。使用工具 ocamlyacc 或 menhir。我还不知道如何使用这两个工具产生友好的错误信息。

多数 yacc 实现都采用了操作符优先级来解决移入/规约冲突，http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html描述了这一解决过程。在实现中我碰到一些典型问题并得以解决。

**解析逗号分割的列表** 考虑表示表达式的非终结符 `expr`，它的其中两个产生式是逗号分割的表达式和分号分割的表达式，约定逗号 (`COMMA`) 优先级高于分号 (`SEMI`)，可以如下实现：

```
expr:
  | ...
  | expr_comma_list %prec below_COMMA { Pexpr_tuple($1) }
  | expr SEMI expr { Pexpr_seq($1,$3) }
  | ...

expr_comma_list:
  | expr COMMA expr_comma_list { $1 :: $3 }
  | expr COMMA expr { [$1; $3] }
```

其中%prec below_COMMA 是指定第一条产生式的优先级，从而在栈为 expr_comma_list 且向前看符号为 COMMA 时，menhir 选择移入而不是规约 expr_comma_list -> expr。

但这种语法描述还有一个问题。当栈内当前为 expr COMMA expr_comma_list 且向前看符号为 SEMI 时，规约 expr_comma_list -> expr 和规约 expr COMMA expr_comma_list -> expr 的优先级均大于 SEMI，均为有效的规约，无法确定选取哪一个，产生规约/规约冲突。

解决方案是把左递归改写为右递归：

```
expr_comma_list:
  | expr_comma_list COMMA expr { $3 :: $1 }
  | expr COMMA expr { [$3; $1] }
```

在使用 expr_comma_list 时要注意反转列表。考虑到函数式语言中在列表头部添加元素比较容易，描述文法时通常用左递归而非右递归，但在上述情形下就不得已采用右递归。OCaml 的语法解析更为复杂，在 parsing/parser.mly 中也能看到一些这样的例子。

### 4.3.3 解析标识符，区分构造器和变量

Caml Light 的语法分析文件中把实现文件划分为用双分号分割的多个 phrase，解析完一个 phrase 后立即进行类型检查、代码生成等工作，并导入全局的值、类型、构造器信息。在解析模式匹配的地方遇到一个标识符时，通过查询之前是否定义过该标识符的构造器来区分该标识符是构造器还是变量。Caml Light 把部分变量是否定义的工作糅合到了解析阶段。

我希望能完整解析源文件后再进行变量定义判断的工作，因此借鉴了 OCaml 的解析器实现，区分大小写两种标识符，大写视为构造器，小写视为变量。

在这里也能一窥语言实现的考量，了解为什么很多支持模式匹配的语言规定构造器使用大写、变量使用小写。

这里也能看到为了语言的一致性，false 和 true 应该看作构造器，使用大写，实现中如果要摈弃 Caml Light 中解析阶段判断是否定义的做法，比较好的方式是在词法分析中把 false 和 true 也作为词法单元。

**可省略的双分号** Caml Light 中要求每一个 phrase(表达式、`let` 定义、`type` 定义或 `exception` 定义) 后面必须跟双分号，而 OCaml 中，非表达式 phrase 前的双分号是可省略的。

方法是区分表达式开始的实现和非表达式开始的实现。我用用一个非终结符 `structure_item` 表示单个非表达式 phrase。

实现文件有三种可能：

- 只包含一个表达式 phrase。
- 以表达式 phrase 开头，之后接双分号。
- 以非表达式 phrase 开头 (用 `structure_tail` 表示)

`structure_tail` 可以看作是实现和表达式开头的实现的差集，有几种可能：

- 空或仅包含双分号
- `SEMISEMI seq_expr structure_tail`
- `SEMISEMI structure_item structure_tail`
- `structure_item structure_tail`，这种情况对应了省略双分号的情形

```
implementation:
  | structure EOF { $1 }

structure:
  | structure_tail { $1 }
  | seq_expr { [make_impl(Pimpl_expr $1)] }
  | seq_expr SEMISEMI structure_tail {
    make_impl(Pimpl_expr $1)::$3 }

structure_tail:
  | /* emtpy */ { [] }
  | SEMISEMI { [] }
  | SEMISEMI seq_expr structure_tail {
    make_impl(Pimpl_expr $2)::$3 }
  | SEMISEMI structure_item structure_tail { $2::$3 }
```

```
  | structure_item structure_tail { $1::$2 }

structure_item:
  | TYPE type_decl_list { make_impl(Pimpl_typedef $2) }
  | LET rec_flag let_binding_list {
    make_impl(Pimpl_letdef($2, $3)) }
  | EXCEPTION constr_decl { make_impl(Pimpl_excdef $2) }
```

**Caml Light 语法分析的一些缺陷** Caml Light 的语法分析文件还存在其他一些缺陷，包括符号优先级定义比较混乱，很多地方应该用`%nonassoc` 而不是`%left` 或`%right` 等。

## 4.4 数据表示

## 4.5 First-class function 的实现

支持 first-class function 需要支持把函数作为值传递，可以存储在变量中、用作返回值，支持内嵌的函数。内嵌函数作为值传递时，它仍可以访问上层函数的局部变量，因此如果作为值传递到外界，需要提供一种方式访问那些变量。

一种方式是闭包，把需要访问的变量和函数代码一起存储。函数代码可以用指向待执行指令的指针表示，而内嵌函数需要访问的上层函数的局部变量就需要保存在一个数据结构里，称为环境。[3] 中提到了一些闭包实现策略的考量。函数应用时没有访问环境，只有构建闭包的代码和访问相应变量的代码需要。在选择环境的表示上我有相当的自由度。

另一种方式是 lambda lifting，可以实现 first-class function 的部分特性。

## 4.6 编译流程概览

- 解析源代码得到抽象语法树
- 类型推导得到带类型的语法树
- 翻译成使用 de Bruijn index 的扩充 $\lambda$-calculus。原始的无类型 $\lambda$-calculus 中唯一的值类型是 $\lambda$ 抽象，该扩充 $\lambda$-calculus 提供了 let、原语操作 (如加法、

比较、创建数组、创建 boxed data 等)。这一部分还包括一个编译模式匹配的子系统。

- 把扩充 $\lambda$-calculus 翻译到 Zinc 抽象机器
- 运行时系统解释执行 Zinc 抽象机器字节码

# 第 5 章　抽象机器

抽象机器 (abstract machine) 一步一步执行程序，但比起具体的机器 (如 x86) 省略了大量的硬件细节。与抽象机器经常混用的一个概念是虚拟机 (virtual machine)。但 virtual machine 这个词被广泛用于操作系统的各层抽象中，因此报告中我用“抽象机器”这个词以与这些概念相区分。

[4] 对抽象机器做了分类。

## 5.1　Strict functional language 的抽象机器

- SECD machine(1964) call-by-value
- Functional Abstract Machine(1983) 是一个扩展、优化的 SECD
- Zinc Abstrace Machine(1990) 可以看作是 Krivine machine 的 call-by-value 版本。

## 5.2　Lazy functional language 的抽象机器

- G-machine(1984)
- Krivine machine(1985)
- Three Instruction Machine(1986)
- Spineless-Tagless G-machine(1989)，用于 Glasgow Haskell Compiler

### 5.2.1　Krivine machine

采用 de Bruijn index 表示的 $\lambda$-calculus 的项形式如下：

$$M ::= n \mid (MN) \mid \lambda M$$

Krivine machine 仅有三条指令：Access、Push 和 Grab，把 call-by-name 的 $\lambda$-calculus 规约为 weak head normal form，它的翻译方案如下：

$$[\![n]\!] = \text{Access}(n)$$

$$
\begin{aligned}
[\![(MN)]\!] &= \text{Push}([\![N]\!]); [\![M]\!] \\
[\![\lambda M]\!] &= \text{Grab}; [\![M]\!]
\end{aligned}
$$

机器有一个代码指针、环境 (一个闭包列表)、栈 (闭包列表)，状态转移方案如下：

表 5.1　Krivine machine

| Code | Env | Stack | Code | Env | Stack |
|---|---|---|---|---|---|
| Access(0); $c$ | $(c_0, e_0) \cdot e$ | $s$ | $c_0$ | $e_0$ | $s$ |
| Access($n+1$); $c$ | $e$ | $s$ | Access($n$) | $e$ | $s$ |
| Push($c'$); $c$ | $e$ | $s$ | $c$ | $e$ | $(c', e) \cdot s$ |
| Grab; $c$ | $e$ | $(c_0, e_0) \cdot s$ | $c$ | $(c_0, e_0) \cdot e$ | $s$ |

### 5.2.2　Three Instruction Machine

是描述 Three Instruction Machine (TIM) 的一个很好的材料。我据此实现了一个简单 lazy functional language 的编译器。TIM 是 spineless 的抽象机器，有三条主要的指令：Take、Push 和 Enter，因此被称为 Three Instruction Machine，可以实现 pure $\lambda$-calculus 的 call-by-name 计算。每条指令其实还有多种地址模式。为了实现算术操作等更多需求，还需要更多的指令。[5] 提供了更多的描述。我对 TIM 的实验并未集成到项目中。

## 5.3　Logic programming language 的抽象机器

常用 Warren Abstract Machine 及其变体。

## 5.4　Zinc 抽象机器

Caml Light 采用了执行方式是编译到 Zinc 抽象机器。

类似于 Krivine machine，Zinc 有代码指针和环境。Zinc 不仅要处理闭包，还要处理其他的值类型，因而引入了一个累加器存放临时结果。Krivine machine 中的栈被拆分成两个栈，参数栈和返回栈。如果不作拆分，翻译方案中将会多出一些交换栈顶元素的操作，会带来性能损失。

处于 tail call 位置的表达式指的是计算完当前表达式后就会执行 `Return`，这样的表达式可能会有优化的翻译方案。下面用 $C$ 表示一般的翻译方案，$T$ 表示处于 tail call 位置的表达式的翻译方案。

### 5.4.1 访问局部变量

$n$ 为 de Bruijn index，翻译方案如下：

$$T[\![n]\!] = C[\![n]\!] = \text{Access}(n)$$

状态转移方案 (前后状态写在两行)：

表 5.2 Zinc: Access

| Code | Acc | Env | Arg stack | Ret stack |
|---|---|---|---|---|
| $\text{Access}(n); c$ | a | $e = v_0 \cdots v_n \cdots$ | $s$ | $r$ |
| $c$ | $v_n$ | $e$ | $s$ | $r$ |

### 5.4.2 应用和 $\lambda$ 抽象

先来看一下 Zinc 的前驱 CAM 是如何翻译函数应用的：

$$\begin{aligned} C[\![n]\!] &= \text{Access}(n) \\ C[\![\lambda M]\!] &= \text{Closure}(C[\![M]\!]; \text{Return}) \\ C[\![M\ N]\!] &= C[\![M]\!]; C[\![N]\!]; \text{Apply} \end{aligned}$$

对于一个 $n$ 参数的函数应用，用 curry 的观点看就是有 $n$ 次单参数的函数应用，将会产生 $n-1$ 个临时闭包。

Zinc 做了一个改进，把所有参数从右到左求值并压入参数栈，把函数放在累加器中，再使用 `Apply` 指令应用到全部 $n$ 个参数上。考虑如下代码：

```
let id x = x in
let f x = id in
let g f x = x in
g (f 3) 4
```

4 压入参数栈后对 `f 3` 求值，3 压栈后应用 `f`，此时栈上的元素是 3 和 4，但提供给 `f` 的参数只有一个。因此我需要一种机制让 `f` 知道它的参数只有一个。一种做法是把参数栈的元素分割开，区分不同函数应用的参数。Zinc 采取的做法是使用 `Pushmark` 指令，在参数栈上压入记号 $\epsilon$。

再考虑一个情况，执行 `Return` 时参数栈顶不一定是 $\epsilon$，可能是其他值，那是和 partial application 相对的情况，即函数应用结果是新函数，新函数要作用在余下的参数上。此时累加器是一个函数，需要把它应用到余下的参数上。

$$C[\![M\ N_1\ N_2 \cdots N_k]\!] = \text{Pushmark}; C[\![N_k]\!]; \text{Push}; \cdots; C[\![N_1]\!]; \text{Push}; C[\![M]\!]; \text{Apply}$$

表 5.3　Zinc: Push, Pushmark, Apply, Return

| Code | Acc | Env | Arg stack | Ret stack |
|---|---|---|---|---|
| Push; $c$ | $a$ | $e$ | $s$ | $r$ |
| $c$ | $a$ | $e$ | $a \cdot s$ | $r$ |
| Pushmark; $c$ | $a$ | $e$ | $s$ | $r$ |
| $c$ | $a$ | $e$ | $\epsilon \cdot s$ | $r$ |
| Apply; $c$ | $a = (c_0, e_0)$ | $e$ | $v \cdot s$ | $r$ |
| $c_0$ | $a$ | $v \cdot e_0$ | $s$ | $(c, e) \cdot r$ |
| Return; $c$ | $a$ | $e$ | $\epsilon \cdot s$ | $(c_0, e_0) \cdot r$ |
| $c_0$ | $a$ | $e_0$ | $s$ | $r$ |
| Return; $c$ | $a = (c_0, e_0)$ | $e$ | $v \cdot s$ | $r$ |
| $c_0$ | $a$ | $e_0$ | $v \cdot s$ | $r$ |

$\lambda$ 抽象作为 first-class 值，在传递时应该使用其闭包表示。它除了被传递外，唯一能做的就是应用到参数上，在函数体执行完之后需要返回到 caller，因此是个 tail call。因为 `Apply` 指令已经把参数栈上的第一个参数移动到环境中了，翻译 $\lambda$ 抽象时可以免去这个操作。结合上述考虑，可以采用如下翻译方案：

$$C[\![\lambda M]\!] = \text{Cur}(T[\![M]\!]; \text{Return})$$

`Cur` 的作用是构建一个闭包。

### 5.4.3　Tail call 翻译方案的优化

再考虑如何优化处于 tail call 位置的应用和 $\lambda$ 抽象的翻译方案。

表 5.4　Zinc: Cur

| Code | Acc | Env | Arg stack | Ret stack |
|---|---|---|---|---|
| Cur($c'$); $c$ | $a$ | $e$ | $s$ | $r$ |
| $c$ | $(c', e)$ | $e$ | $s$ | $r$ |

Tail call 位置的函数应用允许接受过量的参数，此时可看作是接受原本需要数目的参数后返回了一个新函数，新函数应用在了余下的参数上。因此我不需要用 $\epsilon$ 分割参数。翻译 $\lambda$ 抽象时，也不同生成一个闭包等待之后被应用到余下的参数上，可以直接从参数栈上取一个参数移动到环境中，再执行函数体；如果取到 $\epsilon$，就说明参数不够，此时再生成闭包。

$$
\begin{aligned}
T[\![\lambda M]\!] &= \text{Grab}; T[\![M]\!] \\
T[\![M\ N_1\ N_2 \cdots N_k]\!] &= C[\![N_k]\!]; \text{Push}; \cdots; C[\![N_1]\!]; \text{Push}; C[\![M]\!]; \text{Termapply}
\end{aligned}
$$

| Code | Acc | Env | Arg stack | Ret stack |
|---|---|---|---|---|
| Grab; $c$ | $a$ | $e$ | $v \cdot s$ | $r$ |
| $c$ | $a$ | $v \cdot e$ | $s$ | $r$ |
| Grab; $c$ | $a$ | $e$ | $\epsilon \cdot s$ | $(c_0, e_0) \cdot r$ |
| $c_0$ | $(c, e)$ | $e_0$ | $s$ | $r$ |
| Termapply; $c$ | $a = (c_0, e_0)$ | $e$ | $v \cdot s$ | $r$ |
| $c_0$ | $a$ | $v \cdot e_0$ | $s$ | $r$ |

### 5.4.4　`let` 绑定

let $x = N$ in $M$ 可以变换为 $(\lambda M)\ N$ 以消除 `let`，但这么做会引入闭包，更好的方法是直接计算 $N$ 再放到环境中。若有 `and` 引入了多个绑定，只要依次放到环境中即可。对于非 tail call 的情况，`let` 表达式计算完后应该把环境中引入的绑定清除，用 `Endlet` 指令实现。

$$
\begin{aligned}
C[\![\text{let } x_1 = N_1 \text{ and} \ldots \text{and } x_k = N_k \text{ in } M]\!] &= C[\![N_1]\!]; \text{Let}; \ldots C[\![N_k]\!]; \text{Let}; C[\![M]\!]; \text{Endlet k} \\
T[\![\text{let } x_1 = N_1 \text{ and} \ldots \text{and } x_k = N_k \text{ in } M]\!] &= C[\![N_1]\!]; \text{Let}; \ldots C[\![N_k]\!]; \text{Let}; C[\![M]\!] \\
C[\![\text{let rec } x_1 = N_1 \text{ and} \ldots \text{and } x_k = N_k \text{ in } M]\!] &= \text{Dummy } k; C[\![N_1]\!]; \text{Update } k-1; \ldots
\end{aligned}
$$

$$
T[\![\text{let rec } x_1 = N_1 \text{ and} \ldots \text{and } x_k = N_k \text{ in } M]\!] = \begin{array}{l} C[\![N_k]\!]; \text{Update } 0; C[\![M]\!]; \text{Endlet k} \\ \text{Dummy } k; C[\![N_1]\!]; \text{Update } k-1; \ldots \\ C[\![N_k]\!]; \text{Update } 0; C[\![M]\!] \end{array}
$$

| Code | Acc | Env | Arg stack | Ret stack |
|---|---|---|---|---|
| Let; $c$ | $a$ | $e$ | $s$ | $r$ |
| $c$ | $a$ | $a \cdot e$ | $s$ | $r$ |
| Endlet n; $c$ | $a$ | $v_1 \cdots v_n \cdot e$ | $s$ | $r$ |
| $c$ | $a$ | $e$ | $s$ | $r$ |

## 5.5 指令集

下面解释一些主要指令的含义：

- ACCESS $n$ 获取环境中第 $n$ 个槽位，放在累加器中。
- CONSTINT8 $n$ 把 `int8_t` 类型的整数 $n$ 放在累加器中。
- GETGLOBAL $s$ 把槽位为 $s$ 的全局值放在累加器中。
- SETGLOBAL $s$ 把累加器中的值存入槽位 $s$。
- APPLY 函数应用，根据返回地址和当前环境构建闭包压入返回栈，同时把参数栈顶弹出添加到环境中。
- POP 弹出参数栈。
- PUSH 把累加器的值压入参数栈。
- PUSHMARK 把 $\epsilon$ 压入参数栈。
- CUR $l$，用标签 $l$ 指向的代码和当前环境构建闭包，放入累加器中。
- GRAB 若参数栈顶为值，则弹出添加到环境中，否则构建闭包放入累加器并返回。
- TERMAPPLY 函数应用，同时把参数栈顶弹出添加到环境中。
- DUMMY $n$ 在环境中压入 $n$ 个任意值 (内容无关紧要)。
- ENDLET $n$ 在环境中弹出 $n$ 个元素。
- LET 把累加器压入环境。
- UPDATE $n$ 用累加器更新环境的第 $n$ 个值。
- GETFIELD $n$ 累加器更新为其表示的 block 的第 $n$ 个域。
- MAKEBLOCK $tag$, $size$ 根据 tag 和 size 域创建 block。

- SETFIELD 累加器表示的 block 的第 $n$ 个域更新为参数栈中弹出的元素。
- BRANCH $l$ 跳转。
- SWITCH 根据累加器表示的 block 的 tag 和跳转表跳转。
- CCALL1 累加器标示解释器中定义的原语函数，执行该函数。
- RAISE 抛出异常。
- RETURN 若参数栈顶为 $\epsilon$ 则弹出并返回，否则执行累加器中的闭包。

# 第 6 章　数据表示

不同类型语言的实现使用了截然不同的数据表示方式。

对于 Scheme 这样的动态类型语言的实现，数据在运行时需要标志以区分不同类型，所有类型的数据通常需要适配某一种通用的格式，通常是一个字的指针，指向一个 block，block 里存储具体的数据，这种存储方式称为 boxed，记录类型、数组等都是 boxed 类型。对于双精度浮点数，因为占 64 位，而机器字长不足以同时表示指针和所有可能的浮点数，因此浮点数也必须采用 boxed 类型。

因为整数使用的普遍性及性能考虑，整数通常不存储在 block 中并用指针标识，而是用左移一位且最低位置 1 的方式表示。因为指针通常要满足一定的对齐条件，最低位为 0，这种表示方法不会引起歧义。

对于 C、Pascal 这样的 monomorphic 静态类型语言的实现，因为无需进行运行时类型测试，各数据类型不必有统一的格式，因此往往可以有不同的大小。单精度、双精度浮点数可以采取格子的 unboxed 表示，可以有各种大小的整数类型，比如 `int8_t`、`int64_t` 等。不同类型的函数也可以采取不同的调用约定，比如参数为浮点数或整数时，很多架构下在调用函数时会把参数放在不同的寄存器中。

引入 parametric polymorphism 后，问题变得复杂，有三类解决方案：

- 限制 polymorphism 的形式，比如 Modula 中规定抽象类型必须是指针类型，Java 中不允许 unboxed 类型如 `int` 转换为 `Object`。这种方式的缺点是不够直观，并且对语言表现力有较大的牺牲。
- 为不同类型生成多份代码。C++ 模板通常采取这种方式，为不同类型生成不同的代码。缺点是代码大小膨胀，链接复杂性增加，可能需要链接代码生成等。
- 采用 Scheme 式样的处理方式。普遍采用 boxed 类型。缺点是性能损失。

这方面有一些研究成果，比如采取 C 和 Scheme 结合的方法、在 polymorphic 代码中引入类型大小信息等。

## 6.1 Block 和整数的区分

我采取 boxed 类型的方式，多数类型如元组、数组、`float` 等，它们的实际内容都用堆上分配的一块内存表示，使用指针来标识它们，参数传递和变量存储时都使用表示它们的指针。整数则用低位为 1 的值表示。

### 6.1.1 整数

一个整数 $i$ 表示为 $2i+1$，在一个字长为 32 位的机器上 (下面都假定采用 32 位机)，能表示的 `int` 范围不再是 $[-2^{31}, 2^{31})$，丢失了一位变为 $[-2^{30}, 2^{30})$。相应地，整数运算需要做一些修改，比如取反实现为 $2-x$、加法实现为 $x+y-1$ 等。

### 6.1.2 Block

一个 block 由记录元数据的首部和实际存储的数据两部分组成。字符串、数组和闭包的格式有些特殊，其他 block 采用统一的格式，元数据占两个字，第一个字记录 tag、block 的大小和用于垃圾回收的字段 color。第二个字是一个 XOR 链表指针，堆上所有分配的 block 都串在一个 XOR 链表里。

```
bits  31  20 19    8 7   0   31                  0
     +------+-------+-----+ +--------------------+
     | size | color | tag | | xor of prev & next |
     +------+-------+-----+ +--------------------+
```

Tag 占据一个字节，标识了该 block 的类型，可用于区分内置类型 `float`(表示为一个双精度浮点数)、`string`、`array` 和 sum 类型采用了哪一个构造器。注意经过类型检查后，我不需要区分不同代数数据类型，只需要标识特定代数数据类型采用了哪一个构造器。

若 tag 值大于或等于 `No_scan_tag(251)`，就表示这个 block 存储的数据都是不透明的字节信息，不应该把它们解释为指向其他 block 的指针。一个代数数据类型的各个构造器，分别用 tag 值 0、1、2 等表示。`unit` 类型的构造器 `()` 的 tag 值为 1，`bool` 类型的构造器 `false` 的 tag 值为 0，`true` 的 tag 值为 1。

size 域代表 block 的指针域数目，对于 `true` 这类不包含子域的值，其 size 为 0。

color 域用于垃圾回收，对于我采用的 Schorr-Waite graph marking 算法，color 域需要 $\max\lceil \log_2 n + 1 \rceil$ 个 bit，其中 $n$ 是该节点的指针域数目。如果 color 域和 size 域使用相同的长度，如我的实现中所选取的，size 域最大可以为 $2^{12} - 1 = 4095$。元组很难有超过 4095 项，一个类型的构造器数目也很难超过 4095，因此这种方式对于这些类型都是合适的。但是对于字符串长度和数组大小，4095 还是相对较小，因此这两个类型要采取特殊的表示方式，把 color 域的 bit 也作为 size 使用，但这样就需要其他地方存储 color 域，我选择了在 XOR 指针域后再加一个字编码 color。

表6.1整理了一些常见类型值的表示方式：

表 6.1　数据的内部表示

| 值 | 表示 |
|---|---|
| `3` | 3*2+1=7 |
| `'a'` | 97*2+1=195 |
| `true` | 指向一个 tag 为 1，size 为 0 的块的指针 |
| `false` | 指向一个 tag 为 0，size 为 0 的块的指针 |
| `[]` | 指向一个 tag 为 0，size 为 0 的块的指针 |
| `3::[]` | 指向一个 tag 为 1，size 为 2 的块的指针，该块的两个域分别为 2*3+1=7 和一个表示 `[]` 的块 |

### 6.1.3　字符串的表示

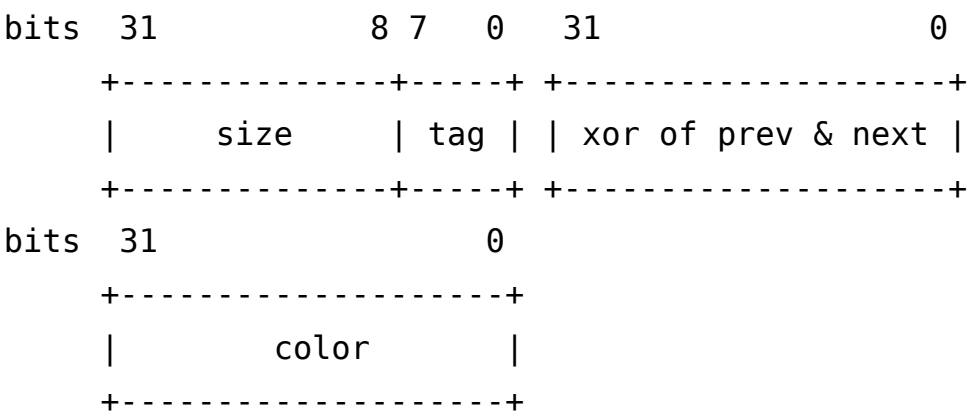
字符串的 block 的 tag 大于 `No_scan_tag`，但元数据存储格式略有区别。因为字符串不包含指向其他节点的指针域，采用 Schorr-Waite graph marking 算法垃圾回收时只需要区分是否被标记过，color 域被缩短为只有一个 bit，其余 bit 都被 size 域占用，size 域共 23 个 bit，最多可以表示长度为 $2^{23} - 1$ 的字符串。

```
bits  31  9  8     8 7   0   31                  0
     +------+-----+-+-----+ +--------------------+
     | size | color | tag | | xor of prev & next |
     +------+-------+-----+ +--------------------+
```

### 6.1.4　数组的表示

数组的 block 的 tag 大于 `No_scan_tag`。原来第一个字的 color 域全部都被 size 域占用，因此最大长度是 $2^{24} - 1$，color 域被移动到 XOR 链表指针域的

下一个字。




### 6.1.5 闭包的表示

闭包的 block 的 tag 小于 `No_scan_tag`，size 域固定为 2，两个域分别为代码指针和表示环境的闭包。

### 6.1.6 环境的表示

为了一致性，环境也用 block 表示，但因为它不会作为值传递，tag 值无关紧要，size 域的值就是环境中存储的变量数。

### 6.1.7 Block 中的 XOR 指针域

使用 mark-sweep 方式的垃圾回收器需要能追踪分配过的堆对象，我在首部的 XOR 指针域中维护了所有分配对象的双链表。对于简单的收集所有对象并标记扫除的垃圾回收器，单链表足矣。但考虑到拓展性，我采用了双链表，以方便未来可能需要支持的操作。因为两个指针域空间开销有点大，我使用了 XOR 链表，即每个节点的指针域为其前驱和后继节点地址的 XOR。根据 XOR 链表头或尾节点的地址即可遍历整个链表。

# 第 7 章　编译、链接和运行

## 7.1　流程概览

解析得到抽象语法树后，之后的编译过程可以拆分为以下阶段：

- 类型检查。检查程序是否合法，类型推导也在这一步进行。
- 把语法树翻译成一种中间表示：扩充的 $\lambda$ – *calculus*
- 把扩充的 $\lambda$ – *calculus* 编译为 Zinc 抽象机器字节码。每个源文件会生成一个目标文件。
- 链接各目标文件得到可被运行时系统执行的字节码文件。
- 运行时系统解释执行。

下面分别介绍各个阶段。

## 7.2　类型检查

解析得到抽象语法树后，进行类型检查判断程序是否合法。项目中使用 Damas-Hindley-Milner 类型系统的 algorithm W 进行类型推导。

Algorithm W 中使用 unification 来判断两个类型是否可能相等，为此需要适配两个类型的形状。比如如果要 unify 函数类型 `int -> a` 和另一个类型 `b -> char`，需要把类型变量 `b` 变为具体类型 `int`、类型变量 `b` 变为具体类型 `int`。

实现 unification 时有两种方法：

- 使用一个类型映射表，记录各个类型变量被映射为什么类型 (具体类型、其他类型变量、或 generic 类型)
- 每个类型变量维护一个可修改的域，表示不相交的集合，类型变量可以是尚未确定与其他类型的关系 (自身为一个集合)，或者被确定与某些类型相同 (与它们在一个集合中)。这里的可变域表示为并查集算法的父节点。

使用可变域性能较高，因此我的实现中也采用了这一方法。

### 7.2.1 类型的 generalization

Damas-Hindley-Milner 类型系统中的 let-polymorphism 阶段，需要把尚未和外部类型联系起来的类型变量 generalize 成 generic 类型。

原始的实现方式是遍历类型 (表示为一棵树)，访问其中所有类型变量，判断是否和外部类型 unify 过，如果没有则 generalize。http://okmij.org/ftp/ML/generalization.html中提到的 Rémy's level-based generalization 采取了一种剪枝策略，也是 Caml Light 实现所采用的。我的项目也实现了这一方法。另外文章中还提到了一种惰性实现，可以优化 occur check，并加速整个类型推导过程的时间复杂度到近似线性。我实现了该方法，但考虑到多数情况下类型树不会很大，occur check 和遍历类型树的代价不会很大，并且惰性方法要引入额外的空间需求，以及实现的复杂性，未在项目中引入。

### 7.2.2 实例化 generic 类型

Caml Light 在实例化 generic 类型时采用了一种技巧 (`cl75/src/compiler/types.ml`)，把实例化分成两个阶段：`copy_type` 和 `cleanup_type`。

`copy_type` 把原来的类型复制了一份，修改了类型树中 generic 类型变量的可变域，指向一个该变量的实例 (不再为 generic)，返回的副本与原本的类型共享那些非 generic 的部分。当原来的类型被复制第二遍的时候，新的副本会与第一份副本共用新生成的非 generic 类型变量。因此对于相同的 generic 类型变量，它的各个副本都会是相同的类型实例。

之后经过 `cleanup_type`，原来的类型被还原成 `copy_type` 前的形态。

`copy_type` 的特性在产生相关联的类型实例时很方便，比如两棵类型树共享一个 generic 的类型变量。那么实例化时希望得到的两个类型树实例的对应类型实例也是相同的。可以这样做：先分别对两棵树进行 `copy_type`，再分别 `cleanup_type`，得到的两个副本保证共用了相同的类型实例。

## 7.3 扩充的 $\lambda$-calculus

ML 的语法比较复杂，在编译到字节码需要进行一些简化，为此 Caml Light 采用了一个扩充的 $\lambda$-calculus 作为中间表示，我也实现了类似的、但更加简化的

扩充 $\lambda$-calculus。

原始的无类型 $\lambda$-calculus 中唯一的值类型是 $\lambda$ 抽象，另外也只有变量和应用两种项形式。经过 de-Bruijn index 的变换后，可以引入环境这一元素 (若干嵌套的 $\lambda$ 抽象中的值的列表)，变量指向环境中特定位置项，因此可以用一个索引值替代，由此我就把 `Lvar of string` 换成了 `Lvar of int`，这里之所以能做这样的变换还得感谢静态作用域。

比如在 `fun x -> fun y -> x` 中，`x` 和 `y` 都在环境中，`x` 可以用 de Bruijn index 1 代替。

在 ML 中，值类型还可以是像 `int` 这样的基本类型，因此我需要一个额外的构造器 `Lconst of constant`。原始的无类型 $\lambda$-calculus 也无法表示加法这样的运算，只能采用 Church encoding。在 ML 这样的实际语言中需要有办法表示这些运算，以及其他很多操作如数组取元素、比较两数、根据构造器和参数创建代数数据类型等，因此我添加了构造器 `Lprim of prim * lambda list`，可以看作 `prim` 表示的原语函数应用在一些参数上。

抛开 let-polymorphism，let 可以等价变形为 $\lambda$ 抽象和应用的一个简单组合 (`let x=... in y` 变换为 `(fun x -> y) (...)`)。而在类型检查通过后，如果把所有值看作是 boxed 类型，并且使用无类型的 $\lambda$-calculus，那么我可以不忽略 let-polymorphism 的影响。但实现为函数应用会有一定的性能开销，因此尽管不是必须有的，我还是提供了构造器 `Llet of lambda list * lambda`。

利用 call-by-value 的求值策略顺序，可以用 $\lambda$ 抽象和应用实现表示顺序执行的`;`(`x; y` 变换为 `(fun _ -> y) x`)，但为了性能考虑设立了 `Lsequence of lambda * lambda` 构造器，它会先执行第一项，再执行第二项。通过使用这种方式，实现了对 C 中语句和表达式的统一。

模式匹配引入了更多的复杂性。当没有模式匹配时，环境中的项 (函数的形式参数) 只有一种引用方式，即视为一个整体引用。但涉及模式匹配后，考虑 `fun ((x,y) as a) ->` ，作为整体的参数可以用 `a` 引用，但模式匹配又引入了构成二元组 `a` 的两个成分 `x`、`y`，它们也可以被函数体引用，所以该如何表示它们呢？另外代数数据类型的构造器可以有多个参数，也可以在模式匹配中出现，需要提供一种办法引用它们的参数。

就像代数数据类型的描述那样 (元组可以看成是提供了语法支持的特殊代数数据类型)，参数可以看成是它们的子域，我使用原语 `Pfield of int` 来

引用某个子域。用于 de Bruijn index 的环境不再是简单变量的列表，列表中的每一项不仅要提供引用参数自身的方法，也要提供方法引用出现在模式匹配中的所有变量。因此我把环境表示为一个列表的列表，每个内层列表代替的原来的简单参数，其中包含所有变量的引用方式。

项目源文件 `front.ml` 实现了抽象语法树到扩充的 $\lambda$-calculus 的翻译。很多节点的翻译很直观：

```
| Pexpr_apply(e,es) ->
    Lapply(go e, List.map go es)
| Pexpr_array es ->
    Lprim(Pmakeblock(0,0), List.map go es)
| Pexpr_constant c ->
    Lconst c
| Pexpr_sequence(e1,e2) ->
    Lsequence(go e1, go e2)
| Pexpr_tuple es ->
    Lprim(Pmakeblock(1,0), List.map go es)
```

比如代表函数应用的 `Pexpr_apply`，代表字面量的 `Pexpr_constant` 等。元组稍复杂一些，可以看作是使用原语 `Pmakeblock` 创建了一个新的 block，其各个域是元组的各个成分。

构造器较为复杂，需要判断在代数数据类型定义中该构造器是否带参数 (`Constr_constant` 或 `Constr_regular`，以及在表达式中是否带参数，选择翻译成一个创建新 block 的原语，或者是个 $\lambda$ 抽象：接受参数以创建 block。

```
| Pexpr_constr(id,arg) ->
    let cd = find_constr_desc id in
    begin match arg with
    | None ->
        begin match cd.info.cs_kind with
        | Constr_constant ->
            Lprim(Pmakeblock cd.info.cs_tag, [])
        | _ ->
            Labstract(Lprim(Pmakeblock cd.info.cs_tag, [Lvar 0]))
        end
    | Some arg ->
        Lprim(Pmakeblock cd.info.cs_tag, [go arg])
    end
```

### 7.3.1 全局值

处于顶层的 `let` 会产生一个全局值 (如基本类型值、函数等)，可以被之后的代码引用。用通常的静态作用域观点看，这样产生的值不在之后代码的作用

域内。当然我可以把在顶层并列放置的 `let` 绑定视为是嵌套的，但这种模型无法解决多文件全局值的引用问题，因此不可行。

解决这个问题需要引入全局值的概念，设置原语 `Pgetglobal of long_ident` 和 `Psetglobal of long_ident` 用于表示读取和写入全局值。

### 7.3.2 翻译模式匹配

模式匹配是最复杂的部分，允许匹配多个代数数据类型复合的结构，混合字面量、变量和构造器，各个匹配项还有顺序要求，因为如果多个模式都能匹配，那么根据语义应该选取最靠前的匹配项。[6] 的 Chapter 5，Philip Wadler 提供了详细的描述。[1] 的 5.2.4 Compiling pattern matching 提供了 Caml Light 中的实现方法。在项目中我又进行了进一步的简化。

对于复杂的模式匹配，比如：

```
match false, false with
| false, true -> ()
| false, false -> ()
```

设想一个按顺序匹配元组的各个项的扩充 $\lambda$-calculus 表示，第一个匹配项的 `false` 匹配成功后，`true` 会匹配失败，此时应该跳转到第二个匹配项。如何实现这样的逻辑呢？

当涉及多个匹配项时，翻译器会把它们组织成 `Lstaticcatch(p1,Lstaticcatch(p2,Lstaticcatch(...)))`，`p1`、`p2` 是各模式匹配项翻译得到的代码，执行 `p1` 时，若发现后续的模式匹配失败了，就会执行 `Lstaticraise`。其效果相当于抛出了一个异常，该异常会被嵌套它的最内层的 `Lstaticcatch` 捕获，尝试执行作为替代的第二条指令。这一机制可以看成是一个静态作用域的异常处理，因此使用了这样的名字：`Lstaticcatch` 和 `Lstaticraise`。

`if` 可以看作是模式匹配的特例，即匹配 `true` 和 `false`，但为了性能考虑，设立了 `Lif` 这一构造器。

### 7.3.3 翻译异常处理

`Lcatch of lambda * lambda` 的第一个参数是 `try` 的代码体，如果抛出异常则把控制权交给模式匹配部分，即第二个参数。

### 7.3.4 扩充 $\lambda$-calculus 定义

`lambda.ml` 中的定义：

```
type lambda =
  | Labstract of lambda
  | Lapply of lambda * lambda list
  | Lcatch of lambda * lambda
  | Lcond of lambda * (constant * lambda) list
  | Lconst of constant
  | Lif of lambda * lambda * lambda
  | Llet of lambda list * lambda
  | Lletrec of lambda list * lambda
  | Lprim of prim * lambda list
  | Lsequence of lambda * lambda
  | Lstaticcatch of lambda * lambda
  | Lstaticraise
  | Lswitch of int * lambda * (constr_tag * lambda) list
  | Lvar of int
```

## 7.4 扩充 $\lambda$-calculus 翻译成 Zinc 抽象机器的 linear code

接下来把中间表示翻译成 Zinc 抽象机器的 linear code。字节码相当于机器码，而 linear code 就相当于汇编代码。Linear code 的定义在文件 `instruction.ml` 中：

```
type zinc_instruction =
  | Kaccess of int
  | Kapply
  | Kbranch of int
  | Kbranchif of int
  | Kbranchifnot of int
  | Kcur of int
  | Kdummy of int
  | Kendlet of int
  | Kgetglobal of long_ident
  | Kgrab
  | Klabel of int
  | Klet
  | Kmakeblock of constr_tag * int
  | Kpoptrap
  | Kprim of prim
  | Kpush
  | Kpushmark
  | Kpushtrap of int
  | Kquote of struct_constant
```

```
| Kreturn
| Ksetglobal of long_ident
| Kswitch of int array
| Ktermapply
| Ktest of bool_test * int
| Kupdate of int
```

`type zinc_instruction` 的多数构造器都和字节码有直观的对应关系，比如构造器 `Kaccess of int` 翻译成字节码时，会额外输出一个 `uint8_t` 类型的操作数。表示的标号构造器 `Klabel of int` 不会生成字节码，但能作为一个标签供其他指令引用。

中间表示翻译成 linear code 的实现在文件 `back.ml` 中，采用了 compile with continuation 的编译方法。其中的主要函数是 `compile_lambda` 中的 `compile_expr : int -> lambda -> instruction list -> instruction list`，第一个参数表示模式匹配失败后需要跳转到的代码，第二个参数是待编译的扩充 $\lambda$-calculus，第三个参数代表该指令执行完后需要执行的 continuation，返回 Zinc 指令的列表 (需要包含 continuation)。

很多结构的翻译是直观的：

```
| Lvar i ->
    Kaccess i :: cont
| Lconst c ->
    Kquote c :: cont
| Lprim(Pdummy n, []) ->
    Kdummy n::cont
| Lprim(Pgetglobal id, []) ->
    Kgetglobal id::cont
```

模式匹配失败后会执行 `Lstaticraise`，翻译的方式是跳转到嵌套该代码的最靠近的 `Lstaticcatch`。

```
| Lstaticraise ->
    Kbranch staticraise::cont
```

有部分结构比如 `Lif`，代码执行路径可能有多条，需要引入一些分支指令。`Lswitch` 根据参数 (必须是 block) 的 tag 值及跳转表进行跳转。

`Lcatch` 的翻译方案是压入一个异常处理栈帧，执行 `try` 代码体时若发生异常，控制权会交给最近的异常处理栈帧，如果它匹配失败就把控制权交给外层的异常处理栈帧。若未发生异常则弹出异常处理栈帧。

### 7.4.1 全局值

扩充 $\lambda$-calculus 中对全局值的处理可以直接翻译成相应的抽象机器指令，同样需要用于处理读取和写入全局值的指令。读取和写入相同名称的指令应该访问同一个内存地址。为了性能的考虑，我不希望在运行时维护一个名称到具体内存地址的映射表。再考虑到一个文件访问另一个文件定义的全局值，我在生成代码后还需要一个过程把这些名字解析到合适的槽位，因此需要下一个阶段——链接。

```
| Lprim(Pgetglobal id, []) ->
    Kgetglobal id::cont
| Lprim(Psetglobal id, [e]) ->
    c_expr e (Ksetglobal id::cont)
```

## 7.5 Linear code 翻译为字节码，生成目标文件

这一阶段的实现在文件 `emit.ml` 中。我使用了一个简单的翻译方案。每个指令拆分为操作码和参数两部分，操作码占据一个字节，参数则有多种形式：`uint8_t`、`int16_t`、标签等。

因为标签定义和引用都是在当前源文件里局部的，生成代码时可以把这些标签引用都解析为字节码中的相对地址。

## 7.6 链接器

正如之前所提到的，这一过程并非必须，但基于性能的考虑需要在编译流程中引入这一阶段。读取各个源文件编译得到的目标文件 (需要重定位)，梳理其中全局值定义和引用的关系，把各条 `Ksetglobal`、`Kgetglobal` 指令解析到正确的槽位是这一阶段的主要目标。

在我的实现中这一部分的代码在 `ld.ml` 中，`ld` 是传统的链接器的名字，我借用了这一名字来贴切地表示这段代码需要完成的任务。

`ld.ml` 使用了一个两遍扫描过程，先扫描各目标文件得到所有的全局值定义并分配槽位，再扫描各目标文件，解析其中的全局值定义和引用，把指令写入可执行文件中。该文件实际上还处理了原语操作。

## 7.7 运行时系统和字节码解释器

运行时系统和字节码解释器用 C 实现，在目录 `runtime/`中。

### 7.7.1 初始化共享的 size 域为 0 的构造器

因为构造器的域不可变，且 size 域为 0 的 block 经常被使用，我可以把 tag 值为 0 到 255 的 256 个 block 缓存起来，供所有 size 域为 0 的 block 共享。Caml Light 的 `cl75/src/runtime/mlvalues.h` 中声明了 `extern header_t first_atoms[];`，`()`、`true` 等就用这些 `first_atoms` 元素的指针表示。我的运行时也采用了这一技巧。

### 7.7.2 全局值初始化

字节码文件中有一些 `float`、`string` 常量，执行时需要一个加载过程，把这些常量分配为 block 供代码使用。另外还有一些 `Psetglobal` 需要引用的槽位，初始化为整数 0(bit pattern 是 1)。

### 7.7.3 指令解码和派发

解释器的主要部分是 `runtime/main.c` 中的一个取指令并解释的循环。常规解释器实现方法是在循环里放置一个 switch 实现指令的解码和派发 (switch-threaded dispatch)：

```
for(;;) {
  switch (*pc++) {
    case ACCESS:
      ...
      break;
    case ADDFLOAT:
      ...
      break;
    ...
  }
}
```

但这样做，指令一条抽象机器指令会引入至少两条 CPU 的跳转指令，即 break 引入的跳转到 switch 处的指令和 switch 跳到某一个标签处的指令。

我可以让 break 实现解码下一条指令并跳转到相应标签处的功能，这样就可以节省一条跳转指令。GCC 提供了 computed goto 特性，可以获取可以获取标签

所指代码的地址 (`void*` 类型)，从而可以手工实现一个跳转表 (token-threaded dispatch)：

```
void *jumptable[] = {&&ACCESS, &&ADDFLOAT, ...};
goto *jumptable[*pc++];

ACCESS:
  ...
  goto *jumptable[*pc++];
ADDFLOAT:
  ...
  goto *jumptable[*pc++];
...
```

http://www.complang.tuwien.ac.at/forth/threaded-code.html有一些关于 threaded code 的讨论。

### 7.7.4 垃圾回收

Caml Light 使用的垃圾回收算法可以参考[7]，结合了 stop-and-copy 和分代 mark-sweep。我的实现则使用了一个非常简单的方法，基于 Schorr-Waite graph marking 算法。该算法不使用栈，只需要在节点上分配极少量信息 ($\lceil \log_2 n + 1 \rceil$ bits，其中 $n$ 是该节点的指针域数目)。在一些资料中被称为 pointer reversal。

该算法的基本思想是访问树的某个节点时，从根到当前节点的链上，对于每一对父子，父节点 $x$ 可以把指向子节点 $y$ 的指针域改成指向 $x$ 的父节点。再使用一个指针 $p$ 表示当前节点的父节点。那么我从 $p$ 出发，沿着链上节点被修改过的指针域，可以回溯到根。每当一棵子树遍历完回到父节点时，需要恢复父节点被修改的指针域，因此需要知道当前在访问哪一个孩子。方法是在节点信息里维护一个域 $s$，表示遍历过程中的访问次数。

触发垃圾回收条件后，把环境、累加器、全局值作为图标记的种子节点，标记完所有可达节点后，用 XOR 链表遍历所有分配的对象，把没有访问过的节点清除。

# 第 8 章　其他

项目中我还实现了一些辅助工具。

## 8.1　Pretty printer

Pretty-print 指的是对文本文件 (特别是标记语言和程序代码) 采用语法高亮、调整换行空格、调整字体等手段使之易于阅读。项目中实现了一个 pretty printer，用于生成进行换行空格排版调整后的代码和调试信息。

下面只考虑进行换行空格排版调整的 pretty printer。Haskell 中的 pretty printer 主要有两大流派，其一是 John Hughes 设计，由 Simon Peyton Jones 修改的，跟随 GHC 一起发行，称为 Hughes-PJ 库；其二是 Philip Wadler 在 The Fun of Programming 中设计的 pretty printer[8]，后由 Leijen 修改，添加了对齐等扩展得到的，称为 Wadler-Leijen 库。

[9] 提出了另一种方法，是对 Wadler/Leijen 的一种改良。原始的 Hughes-PJ 和 Wadler/Leijen 对于并集文档，为了注重性能，使用了贪心算法进行选择。但这样得到的文档不一定是最短的。这一方法以行为单位搜索最短表示。我对它进行了改良，使得能保证返回适应宽度的文档，并且吸收了 Wadler/Leijen 中的 `Nesting` 和 `Column` 构造器，用于提供对齐功能。但实际上这两个构造器的功能能实现更为复杂的功能，比如分栏显示等。

实现分栏显示的关键代码是：

```
let column =
  Column (fun k ->
    if k < 60 then Text (String.make (60-k) ' ')
    else Line true)
```

非常简洁。即使用 `Column` 获取当前列号 `k`，如果 `k<60` 则说明当前行还有空间显示下一个列表元素，不然换行显示。

[10] 提出了一种线性算法,[11] 提供了该算法的不依赖 lazy evaluation 的版本。我实现并扩展了这种方法，引入了对齐操作。

## 8.2 移植到 Web 端

我另外实现了一个代码排版工具和一个可以在浏览器里编译执行的编辑器http://maskray.me/portfolio/caml-featherweight/。

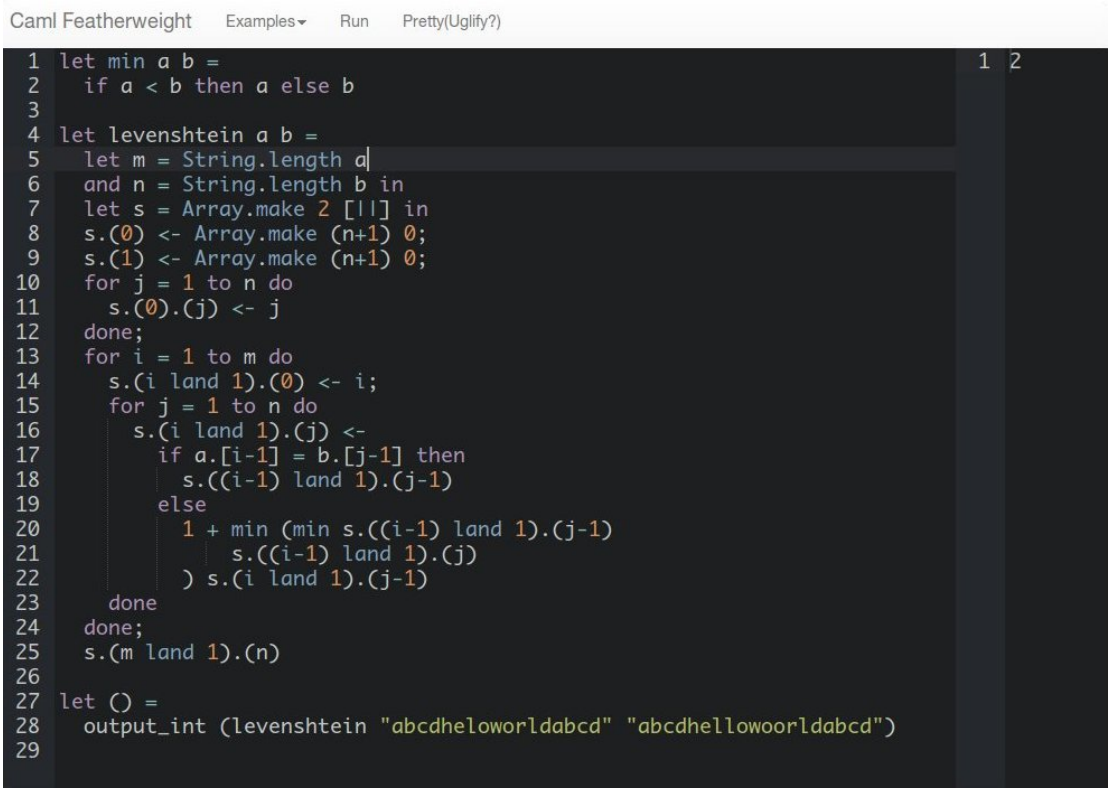



图 8.1 Web 编译器上执行一段示例代码

Caml Featherweight 分为编译器 (OCaml) 和解释器 (C) 两部分，OCaml 部分可以用 Js_of_ocaml 编译为 JavaScript，从而可以在浏览器上执行。emscripten 则可以把 C 编译成 JavaScript。这样就可以在浏览器客户端里实现编译和执行。

### 8.2.1 Js_of_ocaml

Caml Featherweight 在编译生成 phrase index 和链接过程中均需要使用 `seek_out` 来定位，而 Js_of_ocaml 没有实现这一功能，因此需要对 Js_of_ocaml 输出的 JavaScript 代码做一些变更：

- Js_of_ocaml 会把安装目录中的 `runtime.js` 与字节码转成的 JavaScript 拼接在一起，运行时里实现了库函数，而其中的 `seek_out` 虽改变了

channel 的 offset，而控制输出字符的 caml_ml_output 函数并没有利用 offset 信息。

- Js_of_ocaml 得到的 JavaScript 把 OCaml 标准输出和标准出错里的数据用 console.log 和 console.error 输出到 JavaScript 终端里，而我希望数据能显示在 HTML 的一个编辑器框里，因此需要进行重定向。方法是修改 js_print_stdout 和 js_print_stderr 的实现。

```
--- /home/ray/.opam/4.02.1/lib/js_of_ocaml/runtime.js
+++ /home/ray/tmp/runtime.js
@@ -3890,13 +3890,26 @@
         caml_blit_string(buffer,offset,string,0,len);
     }
     var jsstring = string.toString();
-    var id = jsstring.lastIndexOf("\n");
-    if(id < 0)
-        oc.buffer+=jsstring;
-    else {
+    if (oc.fd <= 2) {
+      var id = jsstring.lastIndexOf("\n");
+      if (id < 0)
+        oc.buffer += jsstring;
+      else {
         oc.buffer+=jsstring.substr(0,id+1);
         caml_ml_flush (oc);
         oc.buffer += jsstring.substr(id+1);
+      }
+    } else {
+      var clen = caml_ml_string_length(oc.file.data);
+      if (oc.offset + len <= clen)
+        caml_blit_string(buffer, 0, oc.file.data, oc.offset, len);
+      else {
+        var new_data = caml_create_string(oc.offset+len);
+        caml_blit_string(oc.file.data, 0, new_data, 0, clen);
+        caml_blit_string(buffer, 0, new_data, oc.offset, len);
+        oc.file.data = new_data;
+      }
+      oc.offset += len;
     }
     return 0;
 }
@@ -4313,19 +4326,13 @@
 function js_print_stdout(s) {
   // Do not output the last \n if present
   // as console logging display a newline at the end
-  if(s.charCodeAt(s.length - 1) == 10)
-    s = s.substr(0,s.length - 1 );
-  var v = joo_global_object.console;
```

```
-  v  && v.log && v.log(s);
+  caml_stdout += s;
 }
 //Provides: js_print_stderr
 function js_print_stderr(s) {
   // Do not output the last \n if present
   // as console logging display a newline at the end
-  if(s.charCodeAt(s.length - 1) == 10)
-    s = s.substr(0,s.length - 1 );
-  var v = joo_global_object.console;
-  v && v.error && v.error(s);
+  caml_stdout += s;
 }
 //# 1 "jslib_js_of_ocaml.js"
 // Js_of_ocaml library
```

修改 `_tags` 文件：

```
<*.ml> or "js_main.byte": package(js_of_ocaml),
  package(js_of_ocaml.syntax), syntax(camlp4o)
```

编译并翻译成 JavaScript：

```
cd src
ocamlbuild -use-ocamlfind js_main.byte
js_of_ocaml --no-inline --pretty js_main.byte \
  -o ../build/js/camlfwc.js
```

Js_of_ocaml 实现尚不完善，Web 编译器的行为和本地运行的 Caml Featherweight 有一些不兼容的情况，比如 `Lexing.from_string` 似乎不能用，浮点数小数点的解析也会出错。

### 8.2.2 前端

仿照http://batsh.org/，使用 Bootstrap 3 创建了一个页面，主体是http://ace.c9.io/的两个编辑器框，另外放了几个菜单用于加载样例和运行等。

完整代码在项目目录的 `web/`下，依赖 Node.js、Grunt、Bower、Sinatra。

## 8.3 应用：BCTF 2015 CamlMaze

BCTF 2015 是由蓝莲花战队举办的面向全世界的网络安全夺旗赛 (Capture the Flag)。我在比赛中负责 CamlMaze 的命题工作，CamlMaze 是一道 Caml Featherweight 字节码逆向题。代码中用 Caml Featherweight 实现了迷宫生成，以及和服务端的 socket 通信 (解释器部分做了一些改动)。其中还用到了类似于 Fast Fourier Transform 的 Fast Reed-Muller Transform。

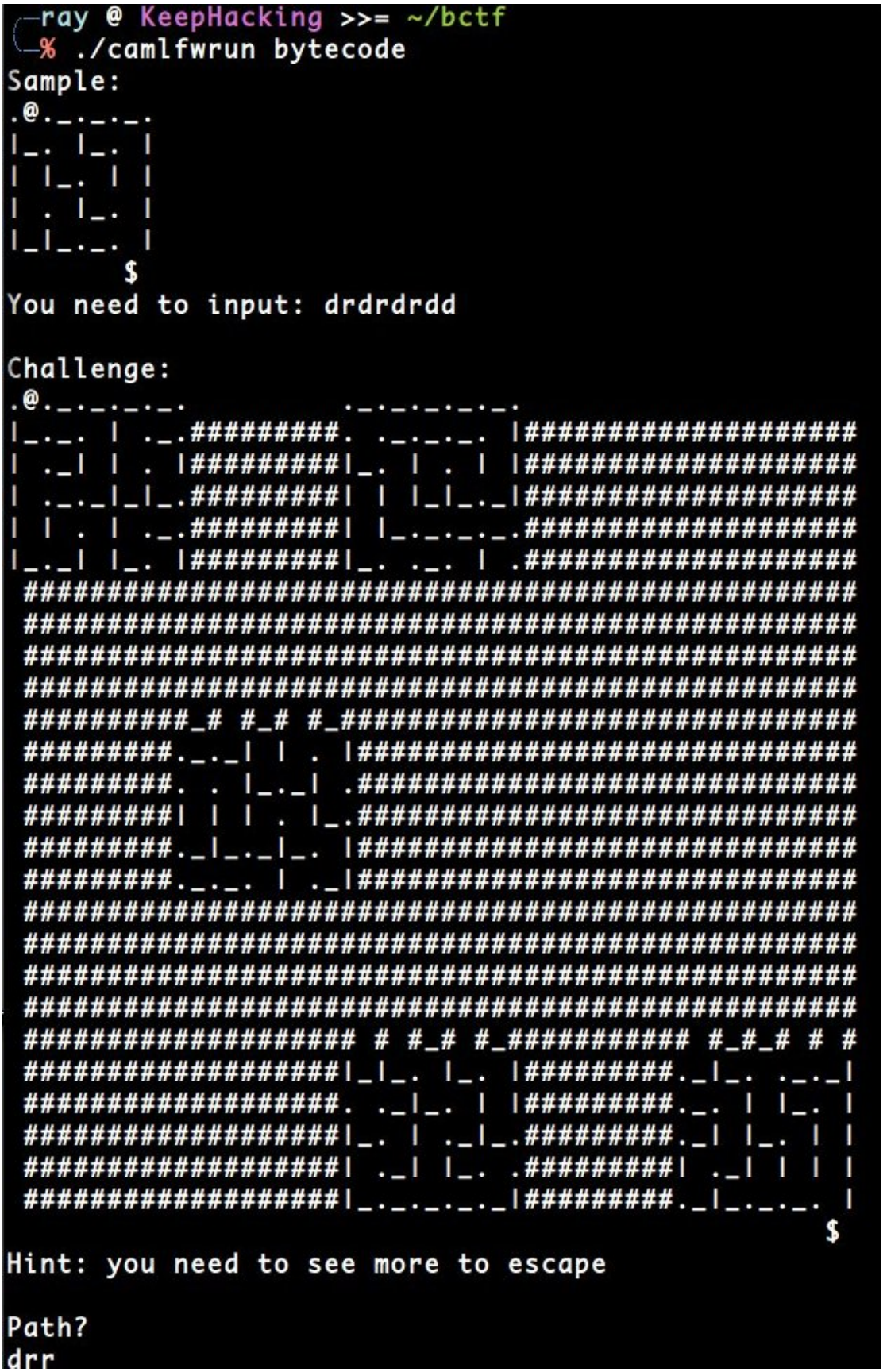



图 8.2　CamlMaze 客户端 (Caml Featherweight 编写)

# 第 9 章　性能

下面把 Caml Featherweight 和 OCaml 的字节码解释器、Python 解释器的性能进行比较。

表 9.1　基本类型

| 名称 | 来源 |
|---|---|
| camlfwrun | Caml Featherweight 的字节码解释器，用 -march=native -O3 -m32 编译 |
| ocamlrun | Arch Linux 的 extra/ocaml 4.02.1-1 包提供的/usr/bin/ocamlrun |
| python | Arch Linux 的 extra/python 3.4.3-2 包提供的/usr/bin/python |

## 9.1　Fast Feed-Muller Transform

下面是用于测试的 OCaml 代码，Caml Featherweight 代码只需把 print_int 修改为 output_int：

```
let reed_muller_transform a =
  for ldm = 18 downto 1 do
    let m = 1 lsl ldm in
    let mh = m lsr 1 in
    let rec go i j =
      if j = mh then
        go (i+m) 0
      else if i < 262144 then (
        a.(i+j+mh) <- a.(i+j) lxor a.(i+j+mh);
        go i (j+1)
      )
    in
    go 0 0
  done

let a = Array.make 262144 0;;
for i = 0 to 262144-1 do
  a.(i) <- i
done;
reed_muller_transform a;
```

```
for i = 0 to 262144-1 do
  if a.(i) <> 0 then
    print_int a.(i)
done
```

用于测试的 Python 代码：

```
#!/usr/bin/env python3
import sys

a = [i for i in range(262144)]

def reed_muller_transform(a):
    for ldm in range(18, 0, -1):
        m = 1 << ldm
        mh = m >> 1
        for i in range(0, 262144, m):
            for j in range(mh):
                a[i+j+mh] ^= a[i+j]

reed_muller_transform(a)
for i in range(262144):
    if a[i]:
        sys.stdout.write(str(a[i]))
```

表 9.2　基本类型

| 名称 | 时间 (秒) |
|---|---|
| camlfwrun | 8.56 |
| ocamlrun | 0.302 |
| python | 1.066 |

可见 Caml Featherweight 在性能上和其他字节码解释器还有巨大差距。

# 第 10 章　收获

## 10.1　创新点

Caml Light、OCaml 是很多科研人员耗费大量时间得到的产品，作为本科生的一个训练项目，我没有那么多时间，也是在学习[1] 的过程中对函数式编译器有了粗浅的了解，在编译技术上面无法做到突破，但在外围的很多东西上做了创新：

- 数据表示和垃圾收集。使用 XOR 链表把对象串接起来，用一个机器字实现了双链表的效果。使用 Schorr-Waite graph marking 算法实现了非常简单的垃圾收集，相关代码只有 100 多行。
- Pretty printer。结合若干 pretty printer 库得到了一个改进的版本，可以方便实现分栏等效果。
- 移植到 Web 端。OCaml 编译到 JavaScript 的库仍不成熟，做了不少改动。

通过一个比较完整的字节码编译器实现，对编译器的流程有了更深入的理解，也更能体会 ML/SML/OCaml 语言设计的一些取舍，有些点也许未必是语言设计者的考虑因素，但确实是从文献和代码中得到的理解。

- SML 库函数较少使用 curry，而大量使用元组组合各个参数。因为 Zinc 之前的很多抽象机器实现 curry 时会产生大量闭包，有性能损失。
- 构造器和变量用大小写区分。这样在模式匹配时可以区分构造器和变量。
- 构造器只接受不超过一个元素，如果超过一个元素则用元组把参数组合起来，且带参数的构造器如果不跟参数单独出现是不合法的，因为它不能被视为函数。Caml Light 的数据表示中接受元组参数的构造器实际上被看作 *n* 元构造器，对于 `A(x,y,z)`，`x,y,z` 是和 `A` 的 tag 在同一个 block 的，而不是存放一个指针域指向 `x,y,z` 形成的元组。
- Caml Light 链接过程中要按依赖顺序指定各个目标文件。如果 Caml Light 在 `Pset_global` 定义符号前先看到了 `Pget_global` 引用，那么将会产生未定义错误。如果要支持向前引用，那么需要进行两边扫描或者为每个符号维护一个引用列表。
- Caml Light 按 phrase 解析并进行代码生成。每一个 phrase 经过了解析、类

型检查、添加到全局表、生成代码的完整过程，因此之前的类型定义给之后阶段的解析提供信息，语法树中可以使用一些语义更丰富的表示，比如出现构造器时可以使用其具体语义信息，而不仅仅是标识符信息。

# 插图索引

# 表格索引

# 参考文献


[1] Leroy X. The zinc experiment: An economical implementation of the ml language, 1990

[2] Chlipala A. Comparing objective caml and standard ml[EB/OL]. (2012-06-08)[2015-06-14]. http://adam.chlipala.net/mlcomp/

[3] Leroy X. Compiling functional languages, 2002

[4] Diehl S, Hartel P, Sestoft P. Abstract machines for programming language implementation, 2000

[5] Sanyal A. Notes on three instruction machine. CS 613: Design and Implementation of Functional Programming Languages

[6] Jones S P. The implementation of funcional programming languages. 1987

[7] Sestoft P. The garbage collector used in caml light, 1994. http://para.inria.fr/~doligez/caml-guts/Sestoft94.txt

[8] Wadler P. A prettier printer, 1998

[9] Bernardy J P. The prettiest printer[EB/OL]. (2012-12-05)[2015-06-14]. http://www.cse.chalmers.se/~bernardy/prettiest.html

[10] Swierstra S D. Linear, online, functional pretty printing, 2004

[11] Swierstra S D, Chitil O. Linear, bounded, functional pretty-printing, 2009

# 致　谢

衷心感谢导师陈文光教授对本人的精心指导。
感谢 THUTHESIS，它让我的论文写作方便了很多。

# 声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

# 附录 A　外文资料综述

## A.1　The Zinc experiment: an econonomical implementation of the ML language

### The Zinc experiment: an econonomical implementation of the ML language

The ML language has grown to a general-purpose, very high-level language particularly well-suited to teaching and fast prototyping. My own experience with the CAML system made me feel the need for yet another implementation of ML, with the following goals in minds:

- to get a small, portable implementation of ML.
- to serve as a testbench for extensions of the language.
- for pedagogical purposes.

ZINC is a recursive acronym ofr “ZINC Is Not CAML”, which is yet another ML implementation.

### Design principles

Modules are a method of decomposing programs which is provided by ZINC. ZINC prefers currying functions to N-ary functions. Traditional evaluation strict functional languages use left-to-right evaluation which does not put up well with multiple application as it will create many temporary closures. To be more efficient, ZINC uses right-left evaluation which is both efficient and transparent for multiple application.

ZINC generate bytecode for an abstract machine, specially designed to fit CAML. Then, the abstract machine gets mapped onto a real architecture is trivial which makes ZINC portable.

### The abstract machine

The ZINC machine can be seen as a Krivine’s machine with marks specialized to call-by-value only, and extended to handle constants as well. The ZAM is equipped with a code pointer and a register holding the current environment. Environments are lists of

values, that is, either closures representing functions, or constants (integers, booleans, $\cdots$). An accumulator has been added to hold intermediate results. The stack has been split into two stacks, one of which is the argument holding arguments to function calls, that is sequences of values, separated by marks. The other stack, the return stack, holds closures, that is pairs of a code pointer and an environment. This design helps reducing stack moves, and allows further refinements in the way environments are represented.

The ZINC machine is specially designed to build less closures. The idea is as follows: when we don't have to build any closures, the current environment does not have to survive the evaluation of the current function body. Therefor we can store it, or part of it, in some volatile location (stack or registers) that will automatically reclaimed when the current function returns.

There consideration has lead us quited naturally to a faily complex abstract machine.

### Data representation

ML data types can be split in three classes:

- atomic, predefined types: "small" integers, floating-point numbers, characters, strings.
- predefined type constructors: functions (closures), vectors, dynamics.
- all other type constructors can be defined in ML, through a very general mechanism called "concrete types".

A value is a 32-bit work which represents either a small integer, or a pointer in the heap. All pointers in the heap are 32-bit aligned, so their low-order bits are always 00. Integers are therefore encoded with a 1 as low-order bit: the integer $n$ is represented by the 32-bit field $\bar{n} = 2n + 1$. The heap is divided in blocks of arbitrary size $n$, preceded by a one-word header. The header consists of:

- 2 bits used by the garbage collector for marking purposes.
- 22 bits holding the size of the block, in words (header excluded)
- 8 bits for the kind of the block, since the GC needs to distinguish between blocks containing valid values, which should be recursively traversed.

Small integers are represented by unboxed, 31-bit integers. Floating-point numbers are allocated in the heap as unstructured blocks of length one, two or three words. Functional values are represented by closures, that is pairs of a code pointer and an environment (a vector, that is a structured block). The code is not allocated in the heap, but in a separate, static zone, without garbage collection. Code blocks are put in the static area, thus they cannot contain pointers to the heap, since the Gc would not know about them. This means that structured constants cannot be at the same time allocated in the heap, and directly put into the code, as immediate operands. Either we allocate them in the static area as well, or we store them in the global table, along with the values of global variables, and access them through one additional indirection.

The sum-of-products allows for a very compact representation of values of concrete types. Not only is the constructor stored in the same tuple as its arguments, but we try to put it in the tag field of the header of the tuple.

Without a record with $n$ fields labeled $l_1, \ldots, l_n$ can be represented by a vector if size $n$, and it is easy to know statically the offset of each label, that is the vector element holding the value associated to this label. With subtyping, we have the additional constraint that the representation of a record with fields $l_1, \ldots, l_n$ must be a valid representation for records whose fields are a subset of $l_1, \ldots, l_n$. THis prevents us form having at the same time a compact representation and static determination of offsets.