Yet Another Functional Language

Marcin Martowicz

1 Introduction

In this document, I provide a brief summary of the compilation pipeline for a standard functional language targeting the LLVM infrastructure. The key features covered include:

- Higher-order functions
- Algebraic data types
- Deep pattern matching

2 Intermediate Representations

2.1 Surface

The high-level AST, directly reflecting the user's syntax.

2.2 Desugared

A desugared version of the Surface IR. Deep pattern matching is compiled into shallow pattern matching to simplify type checking.

2.3 Typed

Each node is annotated with explicit type information.

2.4 ANF

This representation introduces records and C-like type casts. Memory layouts are chosen for algebraic data types and shallow pattern matching is compiled into switch statements. The computations are explicitly ordered by using `let` constructs, resulting in A-Normal Form (ANF). This stage is suitable for optimizations such as function inlining.

2.5 Core

Closure conversion is performed, ensuring that all functions are closed and can be hoisted to the top level. This representation is structured to make translation to LLVM straightforward.

3 Implementation

3.1 Source Position

A neat way to include source position information in the AST nodes is to use a wrapper type:

```
data Node a = Node { pos :: AlexPosn, data :: a }

type Expr = Node ExprData

data ExprData
  = EApp Expr Expr
  ...
```

The Node type definition can be reused across multiple IRs.

3.2 Type Information

Type information from the typechecker could be added to each node using the `Node` type, or explicitly to the constructors' fields. The second option is worth considering, as only a subset of types are relevant for LLVM compilation. However, the first option enables type checking of the intermediate representations.

3.3 Other Information

It is convinient to add an addtional `tag` field to the `Node` type, allowing any other necessary information to be stored using mappings.

```
data Node a = Node { tag :: Tag, data :: a }
```

4 Algebraic Data Types

```
data bst =
  | Leaf
  | Node of bst int bst

let tree = Node(Leaf, 3, Leaf)

let sum (t : bst) : int =
  match t with
  | Leaf -> 0
  | Node(t1, n, t2) -> n + (sum t1) + (sum t2)
```

4.1 Memory Representation

A value of an algebraic data type is a pointer to a block of memory that consists of the constructor tag and the fields of that constructor.

```
let leaf = {0}
let tree = {1, leaf, 3, leaf}
```

4.2 Typing

All values of the same algebraic data type should have the same type. The simplest solution is to hide the fields when a value is created and introduce it back when performing the pattern matching.

```
let leaf = cast *{int} {0}
let tree = cast *{int} {1, leaf, 3, leaf}

let sum (t : *{int}) : int =
  let tag = t#0 in
  switch tag with
  | 0 -> 0
  | 1 ->
    let t' = cast *{int, *{int}, int, *{int}} t
    let t1 = t'#1
    let n  = t'#2
    let t2 = t'#3
    in
    n + (sum t1) + (sum t2)
```

5 Higher-order Functions

```
let add x =
  let f y =
    x + y
  in
  f
...
let add1 = add 1

let dft = 1
let fact n =
  if n == 0 then
    dft
  else
    n * fact (n - 1)
```

5.1 Memory Representation

A value of a function type is a pointer to a block of memory that consists of a function code pointer and an environment that provides the values of free variables of the function. The environment can be represented as the flat sequence of values, or be based on the static links which enables sharing of data. This representation of functions is called a closure, because it closes each function, so that it does not have any free variables. Each function gets one additional parameter which denotes its closure or environment, depending on the design. The advantage of the closure as the parameter is that it can be used for elegant compilation of recursive functions such as `fact`. Each reference to a free variable inside the function is replaced by an explicit lookup in the environment structure.

I pick the approach based on the static links, because it should be easy to implement and it minimizes the memory usage, which is important for a language without garbage collection. Each function allocates its environment at the entry and stores a pointer to the enclosing function's environment in its first field. The variables that are referenced inside nested functions must be saved in this environment.

```
fun f_code f y =
  let env     = f#1
  let cur_env = {env} // not used
  let x       = env#1
  in
  x + y

fun add_code add x =
  let env     = add#1
  let cur_env = {env, x}
  let f       = {f_code, cur_env}
  in
  f

fun some_fun_code ... =
  ...
  let add = {add_code, cur_env}
  ...
  let add_code = add#0
  let add1 = add_code(add, 1)

fun fact_code fact n =
  let env     = fact#1
  let cur_env = {env} // not used
  if n == 0 then
    env#dft
  else
    let fact_code = fact#0 in
    n * fact_code(fact, n - 1)
```

5.2 Typing

It is helpful to split this problem into smaller ones.

5.2.1 Change of The Function Representation

Functions in the Core IR are represented as closures. It suggests that there is an interesting case in the translation of types:

$$[\![\tau_1 \to \tau_2]\!] = *(*([\![\tau_1 \to \tau_2]\!] \times [\![\tau_1]\!] \to [\![\tau_2]\!]) \times \text{env}_{\text{type}})$$

Each function of the type $\tau_1 \to \tau_2$ should get the same type after the translation. To achieve that, we can cast env_{type} to `*void` when creating the closures. Each function can cast its environment argument back to its real type at the entry.

5.2.2 Infinite Parameter Type

In each well-typed program the type of a function is structurally bigger than types of its arguments. In our case it is not true, because each function gets its closure as an argument. If we chose to pass the environment, there is no such problem. To overcome this issue, we can set the type of the closure parameter to be `*void`:

$$[\![\tau_1 \to \tau_2]\!] = *(*(*\text{void} \times [\![\tau_1]\!] \to [\![\tau_2]\!]) \times *\text{void})$$

5.2.3 Summary

Let's compile a function of type $\tau_1 \to \tau_2$ with all needed casts.

```
fun fn_code (fn : *void) (param : [tau_1]) : [tau_2] =
  // 1. Cast the closure parameter to its standard type
  let fn = cast [tau_1 -> tau_2] fn
  // 2. Cast the environment to its real type
  let env = cast enclosing_env_type (fn#1)
  // 3. Create the current environment that can be filled later
  let cur_env = {env, param, 0, ..., 0}
  // 4. Cast the current environment for the closure creation
  let void_cur_env = cast *void cur_env
  ...
  ...
  ...
  // Lookup to the enclosing environment
  let var1 = env#3 + 100
  // Update of the current environment
  cur_env#5 = var1
  ...
  // Closure creation
  let clo = {clo_code, void_cur_env}
  ...
  // Closure application
  let clo_code = clo#0
  let void_clo = cast *void clo
  clo_code(void_clo, arg)
  ...
```

6 Deep Pattern Matching

The standard pattern matching compiler uses the `match` function, which solves the following pattern matching instance problem.

$$\begin{array}{cccccc}
u_1 & u_2 & \cdots & u_m & & \\
\hline
p_{11} & p_{12} & \cdots & p_{1m} & \to & e_1 \\
p_{21} & p_{22} & \cdots & p_{2m} & \to & e_2 \\
\vdots & \vdots & & \vdots & \vdots & \vdots \\
p_{n1} & p_{n2} & \cdots & p_{nm} & \to & e_n
\end{array}$$

The key for efficient compilation is the use of `PatternMatchingSeq(e1, e2)` in the target language. If the pattern match fails in e_1, execution should continue at e_2, which represents the next series of cases to check. It can be expressed by jumping to labels and can be done only in the IR with explicit basic blocks - in our case LLVM.

```
// Surface
match xs, ys with
| []  , []   -> 0
| x:xs, y:ys -> 1

// Desugared
(case xs of
| [] -> case ys of ->
       | [] -> 0
| x:xs -> case ys of
       | y:ys -> 1)
|| PatternMatchingError

// LLVM (simplified)
L0:
  switch xs.tag of
  | 0 -> br L1
  | 1 -> br L2
  | _ -> br L_error
L1:
  switch ys.tag of
  | 0 -> br L_constant0
  | _ -> br L_error
L_constant0:
  br L_merge
L2:
  switch ys.tag of
  | 1 -> br L_constant1
  | _ -> br L_error
L_constant1:
  br L_merge
L_error:
  print "pattern matching error"
  call exit
  br L_merge
L_merge:
  res = phi [(0, L_constant0), (1, L_constant1), (undef, L_error)]
```