02247 Compiler Construction Project Report

Custom Project 2

S204743 - Amirkhon Alimov

AST

High-level idea

Hygge is an expression based language, meaning every program is a single expression which can contain one or more expressions. Therefore, the easiest and simplemest representation for the structure of an expression would be tree, which recursively refers to other `Tree<Expr>`. In the original hyggec compiler provided, an Expr is a discriminated union of different cases of expressions, which have unique constructors and carry data.

Odin implementation

Unlike F#, Odin is a manually managed memory language, so the idea of the AST structure is different, but similar conceptually.

Each expr is just a struct of two fields `type` and `variance`, type reflects an internal compiler type that will be assigned during the typechecking stage, while variance is a C-like union of structs as follows:

```
Expr :: struct {
    type:     ^T,
    variance: ^Expr_Variance,
}
Expr_Variance :: union {
    Sequence,
    Type_Ascription,
    Fun_Decl,
    Fun_App,
    Match_Case,
    Union_Constructor,
    Struct,
    While,
    Type_Decl,
    Let,
    If_Else,
    Assignment,
    Unary_Fun,
    Binary_Fun,
    Variable,
    Field_Access,
    Value,
    Parens,
    Scope,
}
```

Further details about the memory layout, initialization and cleaning up memory are irrelevant and outside the scope of this project, but the reader is welcomed to explore the code base. What's important, is that each `variance` of an expression may contain pointers to different `Expr`, thus expressing recursivity as follows:

```
Sequence :: struct {
    e1: ^Expr,
    e2: ^Expr,
}
```

Lexing/Parsing

Lexing

Lexer passes through a stream of characters and emits/stores tokens which will be used by a parser. Lexing is extremelly trivial so the following structure should be self-explanatory:

```
Lexer :: struct {
    using pos:  Position,
    cursor:     u64,
    data:       []u8,
    tokens:     [dynamic]Token,
}
```

`[dynamic]Token`, just means a dynamic array of Tokens

Parsing

This project implements an optimized top-down recursive descent parser:

```
Parser :: struct {
    cursor      : u64,
    tk_advanced : u64,
    tokens      : [dynamic]Lexer.Token,
}
```

It uses the following Context-Free Grammar, where `ME` is the first rule to match:

```
    ME  ::= AE ( ";" AE )*
    AE  ::= TE ( ( "<-" + "-=" + "*=" + "/=" + "%=" ) TE )?
    TE  ::=
        |   E ( ":" PRETYPE )?
    E   ::=
```

```
        |   "readInt"   "("   ")"
        |   "readFloat" "("   ")"
        |   "print"     "(" AE ")"
        |   "println"   "(" AE ")"
        |   "assert"    "(" AE ")"
        |   "if" D
        |   "let" C
        |   "type" F
        |   "while" B "do" E
        |   "do" AE "while" B
        |   "for" G
        |   "struct" "{" H "}"
        |   "fun" K
        |   B
        |   A
    C   ::=
        |   ( "mutable" )? IDENT ":" PRETYPE "=" AE ";" AE
    D   ::=
        |   B "then" AE "else" AE
    F   ::=
        |   IDENT "=" PRETYPE ";" AE
    G   ::=
        |   "(" AE "," B "," AE ")" AE
    H   ::=
        |   HH ( ";" HH)*
    HH  ::=
        |   ( IDENT "=" AE )?
    K   ::=
        |   "(" L ")" "-" ">" AE
        |   IDENT "(" L ")" ":" PRETYPE "=" AE ";" AE
    L   ::=
        |   ( IDENT ":" PRETYPE ( "," L )* )?
```

This grammar allows to handle complex patterns and operations.

Typechecking

Similarily to the provided hyggec compiler a typechecking system was developed that took into account primitive and composite types. Certain operations like addition or printing or assertation are defined only on a part of the domain of internal types.

Type resolution for following features is:

- ☑ e_{1} / e_{2} arithmetic operation
- ☑ $e_{1}\ %\ e_{2}$ arithmetic operation
- ☑ $max(e_{1},\ e_{2})$ arithmetic operation
- ☑ $min(e_{1},\ e_{2})$ arithmetic operation

All the operations above should resolve in either an integer or a float, where each argument is of that type.

- ☑ $e_{1} <= e_{2}$ relational operation
- ☑ $e_{1} > e_{2}$ relational operation
- ☑ $e_{1} >= e_{2}$ relational operation
- ☑ $e_{1}\ xor\ e_{2}$ relational operation

All the operations above should resolve in either a bool, where each argument is of that type.

- ☑ $x\ +=\ e$ C-style add-assign
- ☑ $x\ -=\ e$ C-style minus-assign
- ☑ $x\ *=\ e$ C-style times-assign
- ☑ $x\ /=\ e$ C-style divide-assign

The computation assignment operations are desugared into a regular assignment and the operation itself, where the e has to resolve int o the same type as the variable x, as such: $$ x\ +=\ e $$ $$ x\ \leftarrow\ x\ +\ e $$

- ☑ "Do..While" Loop

Do While loops are desugared into one iteration of the do block, and then a regular while loop is added sequentially. Similarily to the While loop, the condition has to resolve into a boolean.

- ☑ "For" Loop

For loop is also just a desugared version of the while loop, where initialization is followed by the while loop of the condition and the body of sequence of the for loop body and for loop step.

- ☑ $e_{1}\ sand\ e_{2}$ short-circuit and
- ☑ $e_{1}\ sor\ e_{2}$ short-circuit or

Short-cirtuting sand and sor are desugared into if-else blocks according to boolean logic, both parameters should resolve into booleans.

- ☑ Mutable vs Immutable struct fields A trace immutable fields was kept and upon field access they are checked off against it. Immutable fields cannot be assigned to.

- ☑ Recursive Functions All function calls compile down to jumping to a label, so recursivity is a given.

Intermideate Representation

The intermideate representation is just a structure that can be easily tailored for different target assemblies. Since Hygge is experession-based language it has to be transformed into a statement-based representation which would be run as assembly.

The project does not fully implement the full hygge complitation for all expressions, but the following reasoning of how that would be achieved is presented:

Value/Variable

Whenever, a literal value is presented, it would be allocated in the `.data` section and all following instances of it are to be used as addresses to the memory segment. Meaning a number `42` would be stored under the label of `i_42` and instead of `li` instruction `lw` is used. This approach was chosen over the implementation in hyggec, because it allowed universal referal to values in registers, meaning it was not necessary to distinguish between integer/float literals and respective variables.

Variables are just type dependant loads, either of addresses or words.

Working with floats

As far as I researched, there is no way to simply instantiate floats in memory, so the same approach as hyggec of using the transmute of the float as u32 and then using risc-v instructions to move it to a float register.

In-built functions

For the inbuilt functions simple tailoring is to be made. For example, a `not e` is just `e` transforming into IR block followed by xor the value by itself. Similar approaches would be done for addition and so on.

Min, Max are implemented by desugaring them into an if-else, so it is evaluated through the if method below.

If-Else

Similarily, in if-else blocks, the condition is evaluated first, then its result is compared to 0, if it is 0 it jumps to a corresponding label for the false path, otherwise nothing happens and the true path is evaluated immediately.

Let declaration

Each let declaration is presaved in memory zeroed out, then the block that evaluates it is transformed into IR and the result is stored in memory.

Assignment

Assignment is evaluated by extracting the expression into a variable, which computed just like a declaration. Then the contents of the variable is moved to the variable being assigned to.

Functions

Functions are exracted into their separate label, whilst storing the instruction pointer when they are called, the parameters are either passed by registers or via stack.

Structs

Structs in a simple implementation are not really structs, they could be broken down recursively into multiple variable declarations. Since, memory cache efficiancy is not paramount, then separating fields out into separate variables is not terrible.

If that approach is not appropriate, then structs would be passed via stack.

Loops

Loops are implemented just like functions, where the loop body is the function body and no new parameters are added.