6.110 Design Document

This document outlines our design for the 6.110 Decaf compiler up to phase 3.

Overview

Our compiler is written in Rust and is capable of

- Tokenizing and parsing Decaf programs (phase 1);
- Performing semantic analysis, and emit an SSA-like IR (phase 2);
- Generating unoptimized x86_64 assembly from IR (phase 3);
- Reporting errors clearly in a format similar to the one offered by the Rust compiler;
- Being used as a Godbolt backend!

We will elaborate on the design and implementation of these features below.

Choice of Language

We choose to write our compiler in Rust because Rust has a very powerful type system and a robust borrow checker that we believe will help us eliminate bugs early in the development process. Rust's support for pattern matching and functional-style programming has also proven helpful for writing compilers.

Tokenization (Phase 1)

The tokenizer is implemented by hand and converts the input into a series of whitespace-separated tokens, each associated with a `Span` representing their location in the source code. Errors such as invalid character literals or invalid strings are raised here, and our tokenizer attempts to recover from these errors by ignoring the offending character.

Different types of tokens are represented as variants of the `Token` enum, defined in `src/scan/token.rs`. We adopt a fine-granularity design where every type of token corresponds to a different variant, e.g., we have `Token::Add` and `Token::Sub` as opposed to `Token::Op("+")` and `Token::Op("-")`. We believe that this design helps better leverage Rust's type system and is easier to pattern-match against in downstream processing.

Another noteworthy design of our tokenization step is the association of source location with the tokens. This enables helpful error messages downstream and is achieved with the `Span` data type. Because `Span` is so ubiquitous throughout our compiler, we carefully engineered its representation to take minimal space and to be easily clonable:

```
pub struct Span(Rc<SpanInner>);

struct SpanInner {
    start: Location,
    end: Location,
}

pub struct Location {
    pub source: Rc<Source>,
    pub offset: usize,
```

```
    pub line: usize,
    pub column: usize,
}

pub struct Source {
    pub filename: String,
    pub content: Rc<[char]>, // Guaranteed O(1) indexing and cloning.
}
```

`Span` is immutable and multiple `Span`s can share the same `SpanInner` object with `Rc`. Each `SpanInner` records start (inclusive) and end (exclusive) `Location`s of the span. `Location`s again share the same `Source` through `Rc`, which records both the filename and content of the source file. Content is included in `Source` to make it self-contained and to decouple the abstract notion of "source file" from an OS file, so we don't need to access the file system every time to read the source's content in later stages of our compiler. The source file's content is stored as `Rc<[char]>` as opposed to the more intuitive `String` to allow efficient random access (Rust's `String` uses UTF-8 and does not support random access of Unicode code points by index).

We define a generic type, `Spanned<T>`, to represent "something attached with a `Span`".

```
pub struct Spanned<T> {
    pub inner: T,
    pub span: Span,
}
```

The tokenizer returns a stream of `Spanned<Token>`s.

Parsing (Phase 1)

The output of the tokenizer is converted into an AST following the grammar rules of Decaf. Type-checking is not done at this step; the only rules that are checked are the grammar rules. We do not explicitly construct a parse tree; instead, we directly construct an AST.

We implemented this step by hand using a greedy look-ahead parser. Definitions of AST types can be found in `src/parse/ast.rs`. Locational information is kept in this process by attaching `Span`s to many AST nodes.

For bad inputs, our parser recovers from parsing errors by consuming tokens until hitting a certain set of synchronization tokens (such as semicolons). This simple yet effective strategy allows our parser to report multiple syntax errors for a bad source file.

Semantics Check (Phase 2)

In this step, we type-check our AST and construct a low-level IR. Our low-level IR follows LLVM's design and uses static single-assignment (SSA). That said, our IR is more restrictive than LLVM's IR in that it only allows loads and stores from global variables or predeclared `StackSlot`s (for local and arguments). This is fine in Decaf, as Decaf does not support pointers, and we believe that incorporating this restriction into our IR simplifies our design. Below is the skeleton of our IR (full source at `src/inter/ir.rs`):

```
/// An opaque reference to an instruction
pub struct InstRef(usize);
/// An opaque reference to a block
```

```rust
pub struct BlockRef(usize);
/// An opaque reference to a stack slot.
/// Stack slots are used to represent local variables and function parameters.
pub struct StackSlotRef(usize);
/// An address in memory. This is used for loads and stores.
/// We don't support pointer arithmetic, so we don't need to support arbitrary
/// addresses.
pub enum Address {
    Global(IdentStr),
    Local(StackSlotRef),
}
/// An IR instruction
pub enum Inst {
    Phi(HashMap<BlockRef, InstRef>),
    Add(InstRef, InstRef),
    Sub(InstRef, InstRef),
    // ... More arithmetic, comparison & logical instructions omitted.
    LoadConst(Const),
    Load(Address),
    Store { addr: Address, value: InstRef },
    // Loads and stores to arrays use separate instructions from scalar loads and
    // stores, because they need to take an index and need to be bounds-checked.
    LoadArray { addr: Address, index: InstRef },
    StoreArray { addr: Address, index: InstRef, value: InstRef },
    /// Initialize a stack slot with a value.
    /// Differs from Store in that Initialize works for array too.
    Initialize { stack_slot: StackSlotRef, value: Const },
    Call { method: IdentStr, args: Vec<InstRef> },
    /// Loads a string literal, only used in external calls.
    LoadStringLiteral(String),
}

pub struct Block {
    pub insts: Vec<InstRef>,
    pub term: Terminator,
}

pub struct StackSlot {
    pub ty: Type,
    pub name: Ident,
}

pub enum Terminator {
    Return(Option<InstRef>),
    Jump(BlockRef),
    CondJump { cond: InstRef, true_: BlockRef, false_: BlockRef },
}

pub struct Method {
    pub name: Ident,
    blocks: Vec<Block>,
    insts: Vec<Inst>,
    stack_slots: Vec<StackSlot>,
    pub entry: BlockRef,
    pub return_ty: Type,
    // ... field omitted
```

```
}

pub struct GlobalVar {
    pub name: Ident,
    pub ty: Type,
    pub init: Const,
}

pub struct Program {
    pub imports: HashMap<String, Ident>,
    pub methods: HashMap<String, Method>,
    pub globals: HashMap<String, GlobalVar>,
}
```

Our IR contains a list of methods in a Decaf file. Each method includes a list of Blocks, Instructions, and Stack Slots.

Each Instruction is one "SSA instruction"; for example, `Add(InstRef, InstRef)` is one possible instruction.

Local variables are stored on the stack in a Stack Slot. A Stack Slot represents a certain amount of space reserved in the stack, used to hold the contents of a variable. It roughly corresponds to the `alloca` instruction in LLVM.

A block is a list of instruction references, as well as a Terminator. A block can be thought of as one node in a control-flow graph. An instruction reference points to one instruction in the method. Instructions are stored in the method, not the block, because instructions in one block may reference instructions in another block. The Terminator controls where the program should jump to after the block has finished executing; this can be thought of as the outgoing edge of a node in a control-flow graph.

It's probably possible to convert our low-level IR data structure into LLVM IR, but we have not tested.

As we traverse the AST, we maintain a stack of scopes and symbol tables, and use this to do type-checking. Hierarchical symbol tables based on lexical scopes are transient, and can be represented as in a stack at any point in time. This eliminates the need to maintain parent pointers to symbol tables, which is challenging to do in Rust.

We type-check and build IR both in the same pass. For each statement, the checker & builder function takes the AST statement node and a current block reference, generates code, and returns a block reference to the block with the last IR instruction generated for that statement. Sequential control flow can be generated easily:

```
fn build_stmt(&mut self, stmt: &Stmt, cur_block: BlockRef) -> BlockRef;

fn build_block_no_new_scope(&mut self, block: &Block, mut cur_block: BlockRef)
    -> BlockRef {
    // ...
    for stmt in &block.stmts {
        cur_block = self.build_stmt(stmt, cur_block);
    }
    cur_block
}

fn build_expr(&mut self, expr: &Spanned<Expr>, cur_block: BlockRef)
    -> (BlockRef, InstRef, Type);
```

For an expression, the checker & builder function is similar except that it returns `(BlockRef, InstRef, Type)`, returning `InstRef` allows the value of the expression to be used down the line and the returned `Type` facilitates type checking. If the expression is semantically invalid, an error is emitted to the internal buffer and a special `InstRef`, `InstRef::invalid()`, is returned so the error propagates.

```
fn build_cond(&mut self, expr: &Spanned<Expr>, cur_block: BlockRef,
    true_block: BlockRef, false_block: BlockRef);
```

Things are again a bit different when it comes to conditional expressions due to short-circuit evaluation. We take the suggested approach covered in lecture by supplying the checker & builder function with block references to the true and false branch at the call site.

While our IR has the `phi` instruction, our semantic checker does not emit `phi`s, and the IR is not in strict SSA due to extensive use of stack slots. We expect to introduce `phi`s later using a pass similar to LLVM's `mem2reg`.

Unoptimized Code Generation (Phase 3)

Given our low-level IR, we output x86_64 assembly code. Our low-level IR is already linearized, so this step mostly just has to simply map IR instructions into assembly instructions. Runtime array-out-of-bounds checking, function calling, etc. is implemented in this step.

Booleans are considered to be just as big as ints (8 bytes). The size of `bool` and `int` are controlled by two constants in our program so we may change them later if necessary.

The skeleton for our assembler is shown below. We generate assembly function by function. When generating a function, our code can add assembly instructions to both `.data` and `.text` sections. At the end, all `.data` instructions will be put together and the same goes for `.text`. The merged string is then returned as the full generated assembly.

```
pub struct Assembler {
    program: Program,
    // corresponds to .data
    data: Vec<String>,
    // corresponds to .text
    code: Vec<String>,
}
```

```
impl Assembler {
    pub fn new(program: Program) -> Self { ... }

    // While generating assembly for a method, this method can
    // push new lines of assembly to either .data or .text sections.
    fn assemble_method(&mut self, method: &Method) { ... }

    // Adds some data to .data and label it with the name of the variable.
    fn assemble_global(&mut self, var: &GlobalVar) { ... }

    pub fn assemble(&mut self, file_name: &str) -> String {
        for global in self.program.globals.clone().values() {
            self.assemble_global(global);
        }
        for method in self.program.methods.clone().values() {
            self.assemble_method(method);
        }
        // ... join .data and .text sections
        // together and return the full assembly
    }
}
```

Our current assembly generation is very simple and quite inefficient:

- Every SSA instruction writes its result into a unique location in the stack,
- The function prologues first copies all arguments into their homes on stack,
- We only use `%rax` for most of our assembly code (except function calls). Typically, for an operation, we use `%rax` to store an operand we just read from the stack, and then the next instruction that does the job will read the other operand directly.
- We save all callee-saved registers blindly in function prologues despite us not using any of them.
- Comparison-based jumps are slow because in our IR it is represented in two steps: compare (as `Inst`) and jump (as `Terminator`). Their codes are generated separately. Consequently, the generated code will read the operands from memory, compare with `cmpq`, set result with `setxx`, write the result to memory, read the result back to register, and do a `cmpq` with `$0` and jump with `je` -- that's a lot of instructions and memory traffic and it could have been a simple `cmpq` and `jxx`.

We expect this to improve in phase 4 and 5 as we implement mem2reg and register allocation.

Data flow optimizations (Phase 4)

For phase 4 we implemented three data flow optimizations:

- Global copy propagation
- Dead code elimination
- Global common subexpression elimination / Global value numbering

We will elaborate on these optimizations in this section.

Conversion in and out of SSA form

All of our data flow optimizations operate on SSA forms. While our semantic checker already emits SSA-like IR, it cheated a little bit by generating stack loads and stores for every variable reads and writes, and hence did not insert any phi instructions. Our assembly generation module does not accept phi instructions either. Hence, as a prerequisite to our optimizations, we implemented routines that converts our IR into and out of SSA form.

We convert the IR generated by the semantic checker into SSA form following the algorithm by Cytron el al. In particular, for every variable, phis are inserted at the iterated dominance frontier of blocks containing stores to the variable. We compute the dominance relation of our CFG following "A Simple, Fast Dominance Algorithm" by Cooper et al. In addition, we augment the SSA conversion algorithm by first performing liveness analysis and insert phis of a variable only at blocks it is live-in. Hence, the SSA form emitted by our conversion routine is both **minimal and pruned.**

Just before assembly generation, we implement an SSA-destruction routine that eliminates all phi instructions and convert the IR out of SSA form. We use the standard technique: assign each phi instruction a stack slots, split critical edges, and convert each phi instruction as stores at the end of predecessor blocks and a load at the beginning of the successor block. Because each variable (which used to have one stack slot each) can have multiple phi instructions inserted, and we do not coalesce the stack slots of these phi instructions, the SSA-construction-destruction round trip can cause the function to use more stack space. We consider this de-optimization acceptable for now. This SSA-destruction routine is only temporary and we plan to replace it with a SSA-based register allocator (and rewrite the assembly generator) in phase 5.

An amazing property of SSA form is that for each SSA variable, all uses except those in phi instructions are dominated by a single definition. This effectively reduces a lot of the worklist-based dataflow analysis covered in lectures to DFS's on the dominator tree. As such, **we do not list the dataflow equations we used (as required by the handout)**, because our optimizations, when implemented in SSA form, are not based on dataflow equations and fixed-point worklist algorithm.

Global copy propagation

Thanks to the single-definition-dominates-all-uses property of SSA form, global copy propagation is very easy to perform on SSA form. In pseudo code:

```
another_iteration = true
while another_iteration:
    another_iteration = false
    initialize map src: Map<Var, Var> with the identity map
    for block in dominator tree preorder:
        for inst in block.instructions:
            if inst is "y <- x":
                set src[y] = src[x] and remove inst
            else if inst is "y <= phi(...)":
                if all phi operands are the same, say x:
                    set src[y] = src[x] and remove inst
            else:
                replace each SSA variable x used by inst with src[x]
    for phi in all phi instructions:
        replace each operand x with src[x]
```

```
        if all operands become the same:
            another_iteration = true
```

We note that copy propagation is still iterative and potentially requires more than one traversals of the CFG. This is because the use of a variable in a phi instruction is not dominated by its definition, so a single pre-order traversal does not guarantee that a phi node will be visited after all its predecessors.

Dead code elimination

The single-assignment property of SSA form also greatly simplifies dead code elimination as an SSA variable can only either be entirely useful or entirely useless -- with no middle ground (e.g. interleaving of useful and useless assignments throughout the live range). This reduces the dead code elimination pass to two rules:

- `Call`, `Store`, `StoreArray`, and `Initialize` instructions are useful on their own.
- If an instruction is useful, all instructions / SSA variables it references are useful.

This can obviously be solved with a worklist-like algorithm or DFS. Note how the structure of CFG becomes entirely irrelevant here. Our full dead code elimination pass is implemented less than 40 lines of Rust code.

Global common subexpression elimination

Although phase 4 requires global CSE in the hand out, we actually implemented global value numbering (GVN) instead because they are close enough in SSA world and GVN is more powerful. Our value numbering scheme computes a "canonical form" as a string key for every SSA definition. For commutative operators, our value numbering scheme handles commutativity by always having the lexically smaller operand as the first operand. SSA helps us again here by allowing us to perform the entire optimization as a single DFS pass down the dominator tree, as follows:

```
canonical: Map<Var, String>
stack: Stack<Map<String, Var>>

def dfs_gvn(block):
    for inst in block.instructions:
        c = compute canonical form of variable defined by inst
        canonical[inst] = c
        eliminated = false
        for map in stack from top to bottom:
            if map[c] exists:
                replace inst with a copy from map[c]
                eliminated = true
                break
        if not eliminated:
            stack.top()[c] = inst
    for child with block as immediate dominator:
        stack.push({})
        dfs_gvn(child)
        stack.pop()

dfs_gvn(entry block)
do copy propagation
```

Our GVN replaces common subexpressions with copies, and we immediately perform a copy propagation pass to propagate them.

We note that this approach differs greatly from the available expression analysis covered in lecture. In particular, our GVN scheme will not eliminate common expressions of the following forms:

```
if ...:
    c = a + b
else:
    d = a + b
e = a + b
```

This is because neither the definition of `c` nor `d` dominates that of `e`, although they jointly do. An available expression analysis will show that "a + b" is available immediately before `e`, so a CSE backed by available expressions will be able to optimize the redundant `a + b` away. However, it's hard to do available expression analysis *of the original program* in its transformed SSA form because equivalence in lexical form in the original program can be broken in SSA.

Order of optimizations

We apply the optimizations in the following order:

- Dead code elimination
- Copy propagation
- Common subexpression elimination

All optimizations are implemented in a way that does not assume any particular optimization has been applied. For example, our common subexpression elimination pass can be performed before copy propagation because it does not rely on lexical form but value numbering. We choose to make it the lass pass because it does the most work and having a copy propagation pass that reduces the number of instructions up front will help. Similarly, we choose to run dead code elimination first because it's the simplest and runs in guaranteed linear time.

Phase 5 Outlook

GVN-PRE

To address the limitation of our current GVN algorithm, we plan to implement the GVN-PRE algorithm by VanDrunen and Hosking. PRE stands for Partial Redundancy Elimination and is more powerful than CSE or GVN, which only captures "total redundancy". GVN-PRE would also do loop invariant code motion for free, which is why we hold off writing a separate loop invariant code motion pass. However, GVN-PRE is more complicated than CSE, so we plan to defer its implementation to phase 5.

Constant propagation

We have not yet implemented global constant propagation, but it can be easily incorporated into our GVN pass as follows:

- Evaluate and simplify the canonical form of an arithmetic operation when both operands are constants.

- So far the canonical form of a phi instruction is always the instruction itself. Change it so that when the operands' canonical forms are the same constant the constant gets propagated.
- If the canonical form of an instruction is a constant, replace with it with a `LoadConst` instruction.

We most likely will not need a separate constant propagation pass.

Register allocation

We plan to implement the SSA-based register allocation algorithm as presented by Hack. SSA-based register allocation algorithms are appealing as the interference graph of an SSA-form program is chordal and hence polynomial-time colorable. However, we note that coloring is only a small part of the entire register allocation pipeline. Spilling in the presence of phi instructions may actually be more complicated than in non-SSA programs. Therefore we have the following back up plans:

- If we are unable to faithfully reproduce Hack's algorithm (especially partial spilling), we may choose to adopt a spill-everywhere approach that emits slower code but is easier to implement.
- If that approach turned out to be too hard, we can implement a linear-scan register allocator based on SSA. This is the approach taken by the Go compiler.
- If SSA-based linear scan turned out to be too hard still, we can keep the SSA destruction pass and write a linear scan allocator on the non-SSA program. This is not necessarily an inferior approach as the algorithm is simpler and admits more flexibility. This is also the approach taken by LLVM.

A structured IR after register allocation

Our existing assembly generator directly emits assembly in string form. This makes assembly-level peephole optimizations tricky unless commit to regular expressions (which sounds wobbly and awful). On the other hand, writing an structured IR capturing every possible assembly syntax seems to be too much work and an overkill. We are interested in investigating the possibility of a middle ground -- that is, an structured assembly-level IR that enables useful peephole optimizations yet suppresses the ugly details of assembly, a mixture of Rust `enum` and raw string, probably.

Error Reporting and Diagnosis

Our compiler reports error in a user-friendly manner similar to how the Rust compiler does it:

```
error(parser): unexpected token import
   ⟶ ../public-tests/parser/illegal/illegal-07.dcf:
    |
  1 | void main ( ) {
    |      ---- in method 'main'
    |
  2 |   int import; // import is a reserved word
    |       ^^^^^^ expecting <identifier>
    |
```

Internally, this is achieved with our `Diagonistic` API in `src/utils/diagnostics.rs`. A `Diagnostic` struct represents a diagnostic message, and includes

- A pre-text, e.g., "error(parser): unexpected token import."
- One or more `DiagnosticItem`s. Each `DiagnosticItem` underlines a span in the source file with some color and supplies a short piece of text to the side.
- A post-text, e.g., "(hint:) consider renaming the variable."

A `Diagonostic` struct can be instantiated with a builder-like API and all of our compiler error types implements `To<Diagnostic>`.

Our assembly generation includes `.loc` debugging directives, and we set up a custom Godbolt instance so we can examine the assembly output of any Decaf program.

Difficulties

There are no known difficulties at the moment.

Contributions

Work was done equally by all team members.