Final report

Sebastian Selander
gusselase@student.gu.se

TODOs

Sammanfattning

Colubridae är en kompilator för ett enkelt imperativt programmeringsspråk. I dagsläget kan man skriva program med closures (anonyma funktioner), algebraiska datatyper, loopar, flödeskontroll, variabler, och aritmetik. Tyvärr är inte kompilator felfri. Ett flertal buggar har dykt upp, samt är inte testsviten som existerar idag tillräckligt vältäckande för att kompilatorn ska kunna användas med förtroende.

Utvecklandet av kompilatorn har lärt mig mycket. Följande är en liten lista av saker jag skulle rekommendera till framtida mig, eller andra personer som är intresserade att skapa sitt eget språk.

- Utveckla kompilatorn lite åt gången. Det vill säga, skriva inte parsern för hela programmeringsspråket innan arbetet på kodgeneratorn börjas. Det är istället bättre att ta en liten del av språket och utveckal hela processen för den delen av språket. Exempelvis parser, typechecking, samt kodgenerering för aritmetik och variabler.
- Rekommenderar starkt ett steg i kompilatorn som ger unika namn på alla variabler/funktioner m.m. (känt som name resolution). Det är väldigt skönt att veta att namn är unika.
- Kompilera mot en 'intermediate representation' innan man kompilerar till x86, ARM, eller LLVM. T.ex kan C-stil for-loopar skrivas om till while-loopar, och man behöver därmed bara skriva kodgenerering för while-loopar.
- Man ska inte vara rädd att refaktorisera mycket och tidigt. Jag har själv märkt att det är enkelt att luta sig mot ad-hoc implementationer och det blir inte alls bra i längden. Kompilatorer är komplexa projekt som det är.

! TODO !

Beskriv kortfattat resultatet av ert arbete: kunskaper ni fått, produkten ni skapat.

Produktbeskrivning

Colubridae ser generellt ut som de flesta andra imperativa programeringsspråk, men istället för att skijla på statements och expressions så är allt expressions. Detta tillåter exempelvis

```
let x = { 4 } + if predicate() { 1 } else { 2 }
```

där `predicate` är en funktion som returnerar ett värde av typen `bool`. Värdet av `x` blir då 5 om `predicate()` evaluerar till `true` och annars blir `x == 6`.

Kompilatorn består av fem olika interna representationer:

1. Parsed
2. Renamed
3. Typechecked
4. Lowered
5. LLVM-IR

Parsing

Första steget i kompilatorn är parsing, här omvandlar vi text (ostrukturerad), till ett abstrakt syntaxträd (strukturerat). Nedan är ett exempel på hur resultatet av parsning av en typ av expression ser ut.

```
parse :: String -> Expression
parse s = ...

-- >>> parse "let mut x: int = 2 + 3"
-- Let (Mutable, Just (TyLit Int)) (Ident "x") (BinOp (Lit (IntLit 2)) Add (Lit
(IntLit 3)))
```

Renaming

Andra steget, renaming, där döps variabler om så att varje variabel har ett unikt namn. Exempelvis:

```
//before renaming
type MaybeInt {
    Nothing,
    Just(int),
}
def getNumber(x: MaybeInt) -> int {
    match x {
        Nothing => 0,
        Just(x) => x,
    }
}

//after renaming
type MaybeInt {
    Nothing,
    Just(int),
}
def getNumber(x1: MaybeInt) -> int {
    match x1 {
        Nothing => 0,
        Just(x2) => x2,
    }
}
```

här blir det väldigt tydligt att det är x:et som fås ut från `Just` som menas som uttryck i match-kroppen.

Utöver renaming sker även en till traversering över trädet, här tittar vi att alla funktioner har ett explicit eller implicit return-expression. Traversering säkerställer även att break-expressions endast används inuti loopar.

Typechecking

Typechecking är sista steget i frontenden av kompilatorn. Här säkerställs det att programmet är typkorrekt, det vill säga att man till exempel inte försöker addera ett värde av typen `int` med ett värde av typen `bool`. Utöver detta annoteras även varje expression med dess typ. Typinformationen behövs då LLVM-IR är statiskt och explicit typat.

Desugaring/Lowering

Första steget i kompilatorns backend. Backend börjar här då kompilatorn längre inte ska kunna misslyckas att kompilera programmet. I detta steget sker ett flertal hjälpande förvandlingar. Enklast är väl att förklara med några exempel:

Assign-update

```
//innan
def foo() {
    x += 4;
}

//efter
def foo() {
    x = x + 4;
}
```

Statement-expressions

```
//innan
def foo() -> int {
    3 + { let x = 4; x + x }
}

//efter
def foo() -> int {
    let x1;
    {
        let x = 4;
        x1 = x + x
    }
    3 + x1
}
```

Lambdas/closures

Den mest intressanta och utan tvekan den svåraste förvandling är av lambdas/closures. Här behöver lambdas lyftas till egna funktioner samt fria variabler måste fångas och passeras runt som en closure till funktionen.

```
//innan
def foo() -> int {
    let y = 123;
    // notera att y är en fri variabel i lambdafunktionen
    let f: fn(int) -> int = \x -> x + y;
```

```
    f(1);
}

def f(env: void[], x: int) -> int {
    // metasyntax
    let y = (int) env[0]
    return x + y;
}
//efter
def foo() -> int {
    let y = 123;
    let env: void[] = { y };
    f(env, 1);

}
```

Closuren behöver även heap-allokeras då den passeras över till en annan funktion.

LLVM-IR

Sista steget är att generera LLVM-IR kod. Egentligen det enda viktiga att notera här är att LLVM-IR är i static single-assignment form (SSA).

! TODO !

Beskriv produkten ni skapat. Bifoga gärna skärmdumpar, beskriv designstruktur osv.

! TODO !

Kommentera eventuella alternativa lösningar. Varför valdes de bort under arbetets gång? Sett i efterhand?

Produktkvalitet

! TODO !

Är slutprodukten i er mening stabil eller finns kända brister?

Kunskapsanvändning

! TODO !

Kommentera använd kunskap från föregående kurser. Vad har ni speciellt haft nytta av för moment?

! TODO !

Kommentera eventuella andra informationskällor

Framgångar

Tekniska framgångar

! TODO !

Kommentera vad som har gått bra

Projektarbetet

! TODO !

Kommentera glädjeämnen i projektarbetet

Problem

Tekniska problem

! TODO !

Kommentera ev. problem och hur ni löst dessa

Projektarbetsproblem

! TODO !

Kommentera ev. problem och hur ni löst dessa.

Arbetsplan

! TODO !

Har ni nått ert mål?

! TODO !

Kommentera betydande avvikelser från plan