Final report

Sebastian Selander
gusselase@student.gu.se

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.

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`.

Tanken är att Colubridae ska vara ett programmeringsspråk som lägger tyngd på statisk garanti, det vill säga, om ett problem kan detekteras vid kompileringstid så ska inte programmet kompileras. Användare av Colubridae ska heller inte behöva bry sig om minne i större utsträckning än andra programmeringsspråk som har en garbage collector.

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.

Givet följande uttryck:

```
let mut x: int = 2 + 3
```

ser det ut ungefär så här internt i kompilatorn:

```
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.

```
//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 kroppen av match-case.

Utöver renaming sker även en till traversering över trädet, här tittar vi att alla funktioner har ett explicit eller implicit return-expression. Traverseringen 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.

Lowering

Lowering är första steget i kompilatorns backend. Backenden 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. LLVM-IR skrivs i static single-assignment form (SSA). Det innebär att variabler skrivs till exakt en gång. Följande är ett exempel på hur ett kort Colubridae-program kompileras till LLVM-IR:

```
def main() {
    let mut a = 1;
    a += 1;
    printInt(a + a);
}
```

vilket sen blir:

```
define i1 @printInt(i64 %x) {
    ...
}

define void @main() {
    %named_0 = alloca i64
    store i64 1, i64* %named_0
    %a$1 = alloca i64
    %_0 = load i64, i64* %named_0
    store i64 %_0, i64* %a$1
    %named_1 = alloca i64
    store i64 1, i64* %named_1
    %named_2 = alloca i1
    %_1 = load i64, i64* %a$1
    %_2 = load i64, i64* %named_1
    %_3 = add i64 %_1, %_2
    store i64 %_3, i64* %a$1
    store i64* %a$1, i1* %named_2
    %named_3 = alloca i1
    %_4 = load i64, i64* %a$1
    %_5 = load i64, i64* %a$1
    %_6 = add i64 %_4, %_5
    %_7 = call i1 @printInt(i64 %_6)
    store i1 %_7, i1* %named_3
    ret void
}
```

Produktkvalitet

Trots att det finns ett par buggar i kompilatorn är utvecklingen av den i ett bra stadie. Kodbasen saknar dokumentation, men koden i sig är i relativt god kvalité.

Den mest noterbara buggen just nu dyker upp när man använder en break-expression inuti ett match-case.

```
type Foo {
    Foo,
}
def main() {
    loop {
        match Foo {
            Foo => break,
        }
    }
}
```

Problemet som uppstår är då att LLVM-IRs phi-nod inte får korrekt föregående block som argument. Detta är buggen med högst prioritet.

Kunskapsanvändning

Då det inte är första gången jag utvecklar en kompilator har jag haft mycket nytta av tidigare försök och misslyckanden. Det jag har märkt från tidigare, och tänkt på under utvecklingen av Colubridae

är nyttan av att ha ett bra internt lågnivåspråk som man omvandlar till innan man genererar LLVM-IR.

Denna gången har jag även lagt till några extra tillägg i kompilatorn som jag inte gjort förut har jag tagit nytta av extern information där. Enkelt exempel är strukturen av expressions och statement-expressions, där har jag kikat en del på Rusts grammatik och gjort som det är gjort i där.

I övrigt har jag använt mig av populära implementationstekniker. Två exempel som även är nämnda i planeringsrapporten är Bidirectional typing och Trees that grow. I efterhand vet jag faktiskt inte om jag tyckte valet av att följa Trees that grow var ett produktivt, men det har i alla fall varit lärorikt.

Framgångar

Projektet i helhet har varit en succé. Den bit jag är mest nöjd med är att kompilatorn är relativt bra skriven, och jag kan därmed fortsätta arbeta på den som ett fritidsprojekt.

Problem

Inga större problem har dykit upp under utvecklingen, men självklart har projektet inte gått felfritt. Jag hade lite problem med att implementera closures till higher-order functions, till exempel så var vissa variabler annoterad som om de var vanliga funktioner fast de var higher-order functions, då blev det fel när man försökte anropa dem. Som nämnt tidigare återstår minst ett problem, och det är break-expression i kroppen av ett match-case. Lösningen på detta är antingen att skriva om utan LLVM-IRs phi-nod eller studera phi-noder mer. Personligen föredrar jag att studera phi-noder mer så det är ett av nästa stegen.

Arbetsplan

Arbetet har gått nästan exakt enligt plan. Hann tyvärr inte implementera allt som jag önskat, saknar arrayer och därmed användbara strängar. I helhet är jag nöjd och har uppfyllt de flesta mål jag satt för mig själv.

Slutsats

Projektet har varit lyckat, och jag har uppnått nästan alla de mål jag satte för mig själv. Det har varit otroligt skoj att arbeta på Colubridae under sommaren, och jag kommer utan tvekan fortsätta med projektet på min fritid.