Compiler Construction - Supervision 2

Nandor Licker

October 2019

1 CPS

Consider the following method, defined in an ML-like style:

```
let pow n m =
    if m == 0 then 1
    else n * pow (m - 1)
```

1. Convert the method to CPS.
2. Define an AST which can represent both the CPS and the non-cps forms, expressed as an ML algebraic data type.
3. Show the AST for both forms.
4. Define a method, `to_cps`, which convers the AST to CPS form.
5. Assume arithmetic operators and if-else statements are implemented as methods with continuations. Define an AST which does not have equality and arithmetic expressions, nor if statements. Express the function in it (replace the operators with calls to methods which are assumed to implement them).
6. Build a `to_cps` method which targets the second AST.

2 Past Papers

- 2016 Paper 3 Question 4
- 2015 Paper 3 Question 4

3 Practical

3.1 Missing Returns

Now that the language has `if` statements, explain what happens with the following function:

```
func missing_return(x) {
    if (x == 0) {
        return 1;
    }
}
```

The answer can be found in the emitted code. Is this safe? What can be changed to ensure safety? Is it sensible to continue execution? Modify the relevant part of the code generator, possibly adding methods to the runtime, to handle missing returns safely. Is it possible to detect missing returns at compile time?

3.2 While Loops

Similarly to if statements, implement while loops, along with `break` and `continue`.

```
func while_loop(x) {
    i <- 1;
    j <- 0;
    while (i != x) {
        i <- i + 1;
        if (i == 3) {
            continue;
        }
        j <- j + i;
    }
    return j;
}
```

Implement labelled breaks, similarly to what Java provides - design the syntax such that the while loop's node includes an optional label. Why are such `break` statements safer than arbitrary `goto`s?

3.3 For Loops

Implement for loops, enabling statements of the form:

```
for (i <- 0; i != 5; i <- i + 1) {
    ...
}
```

`break`, with or without labels, along with `continue`, should work in `for` loops as well.

3.4 Fixing Builtin Types

The type system infers the most general polymorphic type for each function in a module, including the prototypes for builtins, such as `input_int`. This is incorrect: because of the polymorphic type, `input_int` can be forced to return any type. Your task is to fix this problem by allowing builtins to be annotated with explicit types. Introduce the following declarations, replacing functions with optional bodies:

```
extern input_int(): Int;
```

Notice that a representation for types in the language syntax is necessary - you are not required to add polymorphic types and type variables, only integers, the unit type and function types.