Compiling Functional Programming Languages
Kesha Hietala (Advisor: Gopalan Nadathur), University of Minnesota Twin Cities

Project Objective
To implement two classical approaches to compiling functional programming languages and to compare their behavior with regard to efficiency.

Functional Programming Languages: What and Why
- A formalism that provides a high-level of abstraction, which allows for:
  - natural support for complex, structured data
  - the ability to treat functions (programs) themselves as data
  - a focus on problem solving rather than machine structure
- A powerful framework for developing complex programs correctly
  - abstraction mechanisms match the conceptual requirements of complex, data-oriented programming
  - mathematical structure facilitates reasoning about programs
  - low level details can be relegated to compilation
- A programming vehicle that is practical and growing in use
  - OCaml, Haskell, F#, and Swift are used in industry and gaining in popularity
  - offer competitive efficiency for all but extremely machine-oriented computations

Approaches to Solving Compilation Problems
- Here we consider two approaches:
  - the Categorical Abstract Machine (CAM), which is the basis for the popular language OCaml and relies on the use of categorical combinators
  - compiling with continuations, which has been used in compilers for the languages Scheme and Standard ML and relies on continuations to make control flow explicit
- Both approaches use closures to associate code with an environment of variable bindings, allowing functions to be treated as first-class objects
- The most significant difference between the two approaches is how they handle control
  - consider code generated for the expression:
    \[
    \begin{array}{l}
    let j = \\
    \quad let y = 3 \\
    \quad in \ let f x = x + y \\
    \quad in \ (f 2) + y
    \end{array}
    \]

CAM Approach
- Evaluate expressions in the context of an environment
- Compile j into something of the following form:
  - \(<\mathrm{bind} \ y \ to \ 3>\)
  - \(<\mathrm{bind} \ f \ to \ a \ closure>\)
  - \(<\mathrm{evaluate} \ (f 2) \ to \ v1>\)
  - \(<\mathrm{evaluate} \ y \ to \ v2>\)
  - \(<\mathrm{apply} \ + \ to \ v1 \ and \ v2>\)
- Requires a machine structure that correctly maintains the environment

Continuations-based Approach
- Isolate where computations should take place next and extract this part into a new let expression
- The binding for j becomes:
  \[
  \begin{array}{l}
  let j = \\
  \quad let y = 3 \\
  \quad in \ let f x = x + y \\
  \quad in \ let w = (f 2) \\
  \quad in \ w + y
  \end{array}
  \]
- Translate the resulting expression into code with no special treatment for control

Problems with Compiling Functional Languages
- Compilation is an essential component to closing the gap between a high-level language and what a machine can understand
- Compiling functional languages poses special difficulties because they treat functions as first-class objects
  - Functions can be returned as values
    \[
    \begin{array}{l}
    fun \ f x = \\
    \quad let \ g y = x + y \\
    \quad in \ g
    \end{array}
    \]
  - Problem: h and 1 must be represented by the same code, but require different values for x
  - Functions can be provided as arguments
    \[
    \begin{array}{l}
    fun \ j = \\
    \quad let \ f x y = x + y \\
    \quad in \ let \ g z = 3 \in \ g \ (f 2)
    \end{array}
    \]
  - Problem: How do we structure the evaluation of g and (f 2) in computing g (f 2)?

Project Achievements
- Developed an understanding of the two different models of compilation
- Implemented both approaches for an expressive fragment of call-by-value functional languages
- Qualitatively characterized differences between the two models relevant to performance
  - in the CAM model the environment must be explicitly managed while in the continuations approach it grows linearly
  - control is built into the instruction sequence in the CAM model whereas explicit transfers are needed in the continuations approach
- e.g. consider the evaluation of the expression: \(\begin{array}{l}
  let \ x = 4 \ in \ ((let \ y = 2 \ in \ y) + x) + 3
  \end{array}\)

CAM Approach
- Start with an empty environment \(e_0\)
- Add \(c_1 : \langle x, 4 \rangle\) to \(e_0\) to obtain \(e_1\)
- Add \(c_2 : \langle y, 2 \rangle\) to \(e_1\) to obtain \(e_2\)
- Evaluate \(y\) to \(v_1\) in \(e_2\)
- Restore \(e_1\)
- Evaluate \(x\) to \(v_0\) in \(e_1\)
- Add \(v_1\) and \(v_2\)

Continuations-based Approach
- Start with an empty environment
- Add \(c_2 : \langle x, 2 \rangle\) to the environment
- Goto \(c_1\)
- \(c_1 : \langle x, 4 \rangle\) to the environment
- Goto \(c_1\)
- \(c_2 : \langle y, 2 \rangle\) to result of \(x * y\)
- Goto \(c_1\)
- \(c_1 \ : \ add \ z \ and \ 3 \ and \ return\)

Current work is attempting to quantify the impact of these differences by running both implementations on large real-world programs.