Compiling Functional Languages

Compilation of functional languages comprises:

- Support for unique features, common design patterns
- Principled techniques for program transformation
- Provably sound optimization methods
What is a Functional Language?

Distinguishing characteristic of functional languages: *functions as first class values*

- Functions treated as program data (values): passed as arguments, returned as results
- Functional languages often called *higher-order* languages
- Program design via *recursion* and *equational reasoning* (instead of looping and hacking)
The language $\lambda_{\text{rec}}$

$x \in Var$  \hspace{1cm} \text{variables}

$l \in Lab$  \hspace{1cm} \text{labels}

$n \in \mathbb{N}$  \hspace{1cm} \text{natural numbers}

$v ::= n \mid \lambda x. e \mid \{l_1 = v_1; \ldots; l_n = v_n\}$  \hspace{1cm} \text{values}

$e ::= x \mid v \mid e + e \mid \text{let } x = e \text{ in } e \mid ee \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.l$  \hspace{1cm} \text{expressions}

- Formal notion of evaluation: $e \rightarrow^* v$
- Compiler should ensure that it implements specification
Examples

\((\lambda x. (x.a + x.b))\{a = 2; b = 5\}) \rightarrow^* 7\) \hspace{1cm} \((\lambda x. \lambda y. x + y)(1)(2) \rightarrow^* 3\)

\((\lambda x. \lambda y. x + y)1 \rightarrow^* \lambda y.1 + y\) \hspace{1cm} \((\lambda f. f1)(\lambda x. x + 1) \rightarrow^* 2\)
Target: Assembler

- Functional paradigm has distinctive, high-level features
- Must still be compiled to assembler:
  - First-order values
  - “Flattened” expressions
  - No free variables in functions
  - Function calls expensive (lots of bookkeeping)
Program Transformation

- Compilation of functional languages viewed as a series of *program transformations*

- Program transformations provably correct
  - Define $e \cong e'$ as $e \rightarrow^* v$ iff $e' \rightarrow^* v$ ($e$, $e'$ may both diverge)
  - Let $[e]$ be a program transformation; then $[e] \cong e$
  - Composition of correct transformations is correct ($\cong$ is transitive)
Eliminating Free Variables

To eliminate free variables in function definitions, closure conversion is used

- Nested functions referring to non-local variables common idiom
- Each function assigned an environment mapping free variables to values
- Free variables turned into local variables, as extra function parameters

\[ \lambda x.x + y \]

becomes

\{ fn = \lambda xx.(xx.arg + xx.env.y); env = \{ y = y \} \}
Definition of Closure Conversion

\[
\begin{align*}
\text{CC}(x) &= x \\
\text{CC}(n) &= n \\
\text{CC}(e_1 + e_2) &= \text{CC}(e_1) + \text{CC}(e_2) \\
\text{CC}([l_1 = e_1; \ldots; l_n = e_n]) &= \{l_1 = \text{CC}(e_1); \ldots; l_n = \text{CC}(e_n)\} \\
\text{CC}(\lambda x. e) &= \begin{cases} 
\text{fn} = \lambda xx. \phi(\text{CC}(e)); \\
\text{env} = \{x_1 = x_1; \ldots; x_n = x_n\} 
\end{cases} \\
\text{where } x, x_1, \ldots, x_n \text{ free variables in } e \text{ and } \\
\phi = [xx.\text{env}.x_1/x_1, \ldots, xx.\text{env}.x_n/x_n, xx.\text{arg}/x] \\
\text{CC}(e_1 e_2) &= \text{let } f = \text{CC}(e_1) \text{ in} \\
& \quad f.\text{fn}([\text{env} = f.\text{env}; \text{arg} = \text{CC}(e_2)]) \\
& \quad \vdots
\end{align*}
\]
Closure Conversion

- Once closure conversion runs on a program, no more non-local variables in function definitions
- Sets stage for later stages of transformation
- $CC(e) \cong e$
A-translation

**A-translation** is the functional version of “flattening”

- Turns tree-structured expressions into linear sequence of expressions
- Takes code a step closer to assembler
- Definable as a function $\text{atrans}$ such that $\text{atrans}(e) \equiv e$

\[
\text{atrans}(2 + (\lambda x. x + 1)(3)) = \begin{array}{l}
\text{let } x_1 = 2 \text{ in }\\
\text{let } x_2 = \lambda x. \text{ let } x_3 = x \text{ in }\\
\text{ let } x_4 = 1 \text{ in }\\
\text{ let } x_5 = x_3 + x_4 \text{ in } x_5\\
\text{ in }\\
\text{ let } x_6 = 3 \text{ in }\\
\text{ let } x_7 = x_2(x_6) \text{ in }\\
\text{ let } x_8 = x_1 + x_7 \text{ in } x_8
\end{array}
\]
Hoisting

Even after A-translation, nested functions may still exist:

\[
\text{atrans}(\lambda x. \lambda y. y) = \\begin{align*}
\text{let } x_1 &= \lambda x. \\text{let } x_2 = (\lambda y. \text{let } x_3 = y \text{ in } x_3) \text{ in } x_2 \\
\text{in } x_1
\end{align*}
\]

*Hoisting* eliminates this nesting, linearizing function definitions:

\[
\text{hoist(}\text{atrans}(\lambda x. \lambda y. y)) = \\begin{align*}
\text{let } x_2 &= \lambda y. \text{let } x_3 = y \text{ in } x_3 \\
\text{let } x_1 &= \lambda x. x_2 \\
\text{in } x_1
\end{align*}
\]

*NB:* Hoisting only works if nested functions contain only local variables; must be performed after closure-conversion
Hoisting Definition

\[
\text{hoist}(e) = \begin{cases} 
\text{if } e = e_1[\lambda x. e_2/f] \text{ and } e_2 \text{ contains no functions} & \text{then let } f = \lambda x. e_2 \text{ in hoist}(e_1) \\
\text{else } e & 
\end{cases}
\]

- Hoisting an expression eliminates nested functions
- \(\text{hoist}(\text{atrans}(e))\) yields linear expression
- \(\text{hoist}(\text{atrans}(\text{CC}(e)))) \equiv e\), series of transformations yielding code very close to assembler
Optimizing

- Functional programming heavy on recursion
- Recursion leads to many function calls
- Function calls are expensive:
  - Register saves
  - Symbol table operations
  - Call stack operations
- Functional programming has been comparatively inefficient historically
Optimizing

- *Tail recursive* style more efficient

- Non-tail recursive factorial in OCaml:

  ```ocaml
  let rec fact x =
    if x = 0 then 1 else x * fact (x - 1)
  ```

- Tail recursive factorial in OCaml:

  ```ocaml
  let fact x =
    let rec tf (x,n) =
      if x = 0 then n else tf(x-1,x*n)
    in tf (x,1)
  ```
Optimizing

• Tail recursive calls don’t need local registers; all computation performed by recursive calls
• Callee function can overwrite caller registers, which don’t need to be saved
• Same call-stack frame can be re-used for recursive call
• Tail recursive computation can be transformed into efficient *iterative* computation (loop)
Continuations

Fact: any recursive computation can be transformed into tail-recursive form

- **Continuation**: a function representing the “rest of the computation”
- In case of factorial, a continuation represents the computation which results after completion of recursive call
Continuation-Passing-Style (CPS)

- *Continuation passing style (CPS)* transforms functions into tail-recursive form, using continuations

- CPS factorial in OCaml, where \( k \) is a continuation:

```ocaml
let fact x =
  let rec cpsf (x,k) =
    if x = 0 then k(1)
    else cpsf(x-1, (fun y -> k(x * y)))
  in cps (x, (fun x -> x))
```
Continuation-Passing-Style (CPS)

- CPS transformation can be defined as a general transformation technique
- $\text{CPS}(e) \cong e$
  - Incredibly difficult proof, long research effort
  - Used in compilers before proven correct
The Big Picture

- We can now chain our series of transformations:
  - \( \text{hoist(atrans(CC(CPS(e))))} \cong e \), close to assembler

- All transformations preserve well-defined semantics, bring us close to assembler
The Big Picture

- In the traditional compiler (pioneered by Lisp community at MIT):
  1. Lexical analysis
  2. Parsing
  3. Static type analysis
  4. IR transformation ($\lambda_{rec}$-type language)
  5. CPS transform
  6. CPS optimization
  7. Closure conversion
  8. A-translation
  9. Hoisting
  10. Instruction selection and register allocation
  11. Code generation
Other Optimizations

- **Inline expansion**: replacing a function call with the expanded function body
- **Dead function removal**: unused function definitions (due to verbosity or inline expansion) can be eliminated
- **Loop invariant hoisting**: turning recursive function calls into loops via CPS transformation opens possibility of loop optimizations