Functional Language Interpreters

Ming-Hwa Wang, Ph.D.
COEN 357 Code Generation and Optimization
Department of Computer Engineering
Santa Clara University

Functional Language Issues

- first-order pure functional programming languages support equational reasoning, function without side effects, no higher-order functions, no assignments (immutable variables)
- higher-order functional languages with nested lexical scope (or block structure) and higher-order functions (allow functions to be passed as arguments to other functions, or returned as results.)
- imperative programming languages can have side effects
- lazy evaluation and eager evaluation

Implementation

- In language without nested functions, the run-time representation of a function value can be the address of the machine code for that function. This address can be passed as an argument, stored in a variable, etc., when it is time to call the function, the address is loaded into a register, and the "call to address contained in register" instruction is used.
- For a languages with nested functions, it represents a function-variable as closure: a record that contains the machine-code pointer and a way to access the necessary non-local variables (the environment.)
  - stack-allocated activation records with static link: it takes a chain of pointer dereferences to get to the outermost variables, and the garbage collector cannot collect the intermediate links along this chain even if the program is going to use only the outermost variables
  - heap-allocated activation records: let the garbage collect to determine if it is safe to reclaim the frame (only save escaping-variable: that are used by inner-nested functions)
- Since a pure functional language can't have assignments, it has to produce new data structures instead of updating old ones. Use continuation-based I/O to express input/output: function returns result.

Optimization of Pure Functional Languages

- functional-language compilers can make use of the same kinds of optimizations as imperative-language
- pure functional language has no side effect, so substitution always works
- inline expansion: replace a function call with a copy of the function body
  - avoid variable capture
    - rename, or α-convert, the formal parameters before substitution
    - rename the actual parameters before substitution
    - an earlier pass to rename all variables so that the same variable-name is never declared twice (best solution)
  - rules for inlining: when the actual parameter is a nontrivial expression, we must first assign it to a new variable (thus we may not need to recomputed)
  - dead function elimination: if all the calls to a function is inline expanded, and if the function is not passed as an argument or referenced in any other way, it is eliminated
  - inlining recursive function using the loop-preheader transformation: split the recursive funcation into two functions, a prelude called from outside, and a loop headers called from inside; every call to the loop header will be a recursive call from within itself, except for a single call from the prelude
  - loop-invariant hoisting: replace every use of invariant variable by the value
  - cascading inlining
  - avoiding code explosion:
    - expand only those function-call sites that are very frequently executed; the frequency can be determined by static estimation (loop-nest depth) or feedback from an execution profiler
    - expand functions with very small bodies
    - expand functions called only once
- Closure conversion
  - the closure conversion phase translates the program so that none of the functions appears to access free (non-local) variables by turning each free-variable access into a formal parameter access
  - function values are represented as closure: when the programmer passes a function as an argument, the compiler should pass the code pointer and environment as two adjacent arguments
  - unknown types of static links in closure: use type-checking
- Efficient tail recursion: tail recursion elimination by jump
- Lazy evaluation or call-by-need
  - β-substitution: if $f(x) = B$ with some function body B then any application of $f(E)$ to an expression E is equivalent to B with every occurrence of x replaced by E
  - a program compiled with lazy evaluation will not evaluate any expression unless its value is demanded by some other part of the computation
  - call-by-name: each variable is not a single value, but is a thunk (a function that computes the value on demand) that each thunk may be executed many times, each time redundantly yields the same value; at each place where a variable is created, the compiler creates a function value, and everywhere a variable is used, the compiler puts a function application
  - call-by-need: a modification of call-by-name that never evaluate the same thunk twice; each thunk is equipped with a memo slot to store the value
  - lazy evaluation can do optimizations that strict functional or imperative compiler can't not: invariant hosting, dead-code elimination, and deforestation (removes intermediate lists and trees and does everything in one pass)

- strictness analysis: the overhead of thunk is high, put thunk only where they are needed
  - definition of strictness: a function *f(x)* is strict in *x* if, whenever some actual parameter *a* would failed to terminate, then *f(a)* would also failed to terminate
  - approximate strictness analysis: exact strictness analysis is not computable, thus compiler must use a conservative approximation; where the exact strictness of a function argument cannot be determined, the argument must be assume nonstrict; then a thunk will be created for it