Algebraic Types for Closure Conversion and Analysis

Andrew Tolmach

Types in Compilation Workshop

Amsterdam

June 8, 1997

We want you!

- To work on an exciting compiler project
- Domain-specific language design and implementation
- Functional-style intermediate languages
- Semantics-directed compiling
- Opportunity to field real systems
- There's just one catch...

Ada. What an interesting language!

- Strongly typed
- No function pointers at all (in Ada83)
- An opportunity
  - to explore strongly-typed compilation pipeline
  - to re-examine how to implement higher-order functions
- Result: a simple approach to typed closure conversion that generates a useful closure analysis "for free"

Outline

- Typed closure conversion
- Algebraic types and interpretation
- Closure analysis and optimization
- Restrictions:
  - Whole program assumed available
  - Program is monomorphic

Typing Closures

- Closures are normally represented as records containing code pointers and free variables.
- In "closure-passing" style [Appel&Jim89], express closure construction and application in terms of ordinary record operations.
  - Backend Simplicity
  - Leverage Optimization
- But two functions with the same type (e.g., `int->int`) may have quite different sets of free variables, so the (structural) types of their closure records are incompatible, e.g.

```
val f = if b then fn x => x + 1 else fn x => x + a
```

- Cannot just perform closure conversion and expect the resulting program to be well-typed.

Previous approaches

- [Minamide, Morrisett, Harper96]
- Add existential (or translucent) types; or
- Add special-purpose closure construction and application operators, with special type rules
- Either approach adds complexity to system
- Existing, standard optimizations cannot be used directly on closure-converted code
- (Still have code pointers)

Interpretive Dispatch

- Only a finite number of functions are defined in the whole program, and every code pointer must point to one of them
- So can replace code pointer in closure by the (unique) name of the function; and
- Replace closure applications by calls to a dispatch function that
  - does a case dispatch on the name in the closure, and
  - invokes the corresponding function
- Idea (in untyped settings) due to [Reynolds72, Warren82]
- Need whole program in order to build dispatch function

Algebraic Types for Closures

- In typed setting, a given closure application site can only apply closures of a particular type
- To make the interpretive dispatch type-correct (in monomorphic setting), we need multiple dispatch functions, one for each (source) program function type
- The set of closures that might be passed to a given dispatch function form an ordinary algebraic type:
  - function names are data constructors
  - free variables are associated data values
- No special type machinery is needed to describe converted program
- [Bell, Bellegarde, and Hook97]

Example: `twice` function

SOURCE CODE

```
fun twice(i:int,f:int->int) = f(f i)
fun add(y:int) = let fun a(x:int) = x + y in a end
fun mul(y:int) = let fun m(x:int) = x * y in m end
fun doit(b:bool,i:int) =
   if b then twice(i,add 1) else twice(i,mul 3)
```

AFTER CLOSURE CONVERSION

```
datatype clos = A of int | M of int
fun dispatch(A y,x) = a'(x,y)
  | dispatch(M y,x) = m'(x,y)
and a'(x,y) = x + y      and m'(x,y) = x * y
fun twice(i:int,f:clos) = dispatch(f,dispatch(f,i))
fun add(y:int) = A y    fun mul(y:int) = M y
fun doit(b:bool,i:int) =
  if b then twice(i,add 1) else twice(i,mul 3)
```

Multiple Closure Datatypes (I)

SOURCE CODE

```
fun twice (i:int) : (int->int)->int =
    let fun t(f:int->int) = f(f i) in t end
fun add (y:int) = let fun a (x:int) = x + y in a end
fun mul (y:int) = let fun m (x:int) = x * y in m end
fun doit (b:bool,i:int) =
   let val u:(int->int)->int = twice i
   in if b then u(add 1) else u(mul 3) end
```

Multiple Closure Datatypes (II)

AFTER CLOSURE CONVERSION

```
datatype tclos = T of int  (* (int->int)->int *)
and      iclos = A of int | B of int   (* int->int *)
fun idispatch(A y,x) = a’(x,y)
  | idispatch(M y,x) = m’(x,y)
and tdispatch(T i,f) = t’(f,i)
and a’(x,y) = x + y      and m’(x,y) = x * y
and t’(f,i) = idispatch(f,idispatch(f,i))
fun twice (i:int) = T i
fun add (y:int) = A y    fun mul (y:int) = M y
fun doit (b:bool,i:int) =
   let val u:tclos = twice i
   in if b then tdispatch(u,add 1)
           else tdispatch(u,mul 3)
   end
```

Recursive Closures : Integer Sets

SOURCE CODE

```
datatype env = Env of int -> bool
val empty : env = let fun f i = false in Env f end
fun insert(Env e,k) : env =
  let fun g i = if i = k then true else e i in Env g end
fun lookup(Env e,k) : bool = e k
```

AFTER CLOSURE CONVERSION

```
datatype clos = F | G of int * clos
and env = Env of clos
fun dispatch(F,i) = f' i
  | dispatch(G(k,e),i) = g'(i,k,e)
and f' i = false
and g'(i,k,e) = if i = k then true else dispatch(e,i)
val empty = Env F
fun insert(Env e,k) = Env(G(e,k))
fun lookup(Env e,k) = dispatch(e,k)
```

Outline of transformation

- Each distinct arrow type is converted to a unique corresponding closure datatype.
- A "lambda-lifted" variant is introduced for each function definition, by making the free vars into explicit args; this will be invoked by the `dispatch` function.
- Each function definition is replaced by a binding to the application of a fresh closure constructor to the free vars.
- Calls to unknown functions become calls to the appropriate `dispatch` function, passing the closure value as an extra argument.
- Finally, closure datatype definitions and `dispatch` functions are constructed for the whole program.

Pros & Cons of Interpretation

- After closure conversion, each unknown function call now requires
  - a (known) call to the `dispatch` function
  - a case evaluation
  - a (known) call to a lifted function
- (Since only case over closure constructor is in `dispatch` function, could use code pointers of lifted functions as closure data constructor tags. Then, at last moment, could drop types and convert case back to indirect jump.)
- But simple, general-purpose optimizations are very effective on first-order code!
- Datatypes and dispatch functions encode a simple closure analysis that can drive optimizer.

Standard Optimizations

- Inlining of small or called-once functions
  - Conservative: guaranteed wins
  - (Aggressive: heuristic-based)
- Dead-code elimination
- Propagation of constants and constructed values
- Simplification of case expressions over known values
- Usual representation tricks for datatypes
- Unboxed representations for small, non-recursive datatypes
  - Including sum types
- (Specialization of functions on common arguments)

Uncurrying

- Optimizing closure-converted code gives uncurrying "for free"

```
fun f(x1) = let fun f2(x2) = e in f2 end
val a = ((f 1) 2)
```

```
datatype clos = Cf2 of int | ...  (* int -> int *)
fun dispatch(c:clos,x2) =
   case c of
     Cf2 x1 => f2'(x2,x1)
   | ...
and f2'(x2,x1) = e
fun f(x1) = Cf2(x1)
val a = dispatch(f 1,2)
```

```
val a = f2'(2,1)
```

Opportunities for Inlining

- In lucky cases, a closure datatype might have only one constructor, e.g.

```
datatype tclos = T of int
```

- Or, argument to dispatch function might become known
- Then dispatch function can be inlined at call site, producing a direct call
  - Standard, conservative optimizer does this with no extra bother
  - Direct call may itself be suitable for inlining
- If closure datatype has just a few constructors, an aggressive optimizer might still choose to inline dispatch function

Optimized closure representations

- If a closure must be built, can choose optimized representation because know all the possible constructors.
- Standard datatype representation tricks will avoid building heap-allocated closures for constructors with no free variables, and won't waste tag space unnecessarily on the other constructors

```
datatype clos = F | G of int * clos
and env = Env of clos
```

- If closure datatype is non-recursive and no constructor has too many free vars, can use unboxed representation for closure values.

```
datatype clos = A of int | M of int
```

Making finer type distinctions

- The fewer constructors a closure datatype has, the more likely it is to benefit from these sorts of optimizations.
- Basic division into structural types is dictated by choice of algorithm and data structures.
- User code may make finer distinctions, however, e.g.

```
datatype dollars = Dollars of int
datatype guilders = Guilders of int
```

- Usual aim is to have type-checker help enforce internal consistency of program.
- But also get finer distinctions among functions, e.g.,

```
f: dollars -> guilders     g: guilders -> dollars
```

- Possible performance benefit.

Explicit closure analysis

- Compiler can perform more precise closure analysis
  - Sestoft, Bondorf, Shivers, Heintze, etc.
- When can results of such an analysis be recorded in our framework?
- Answer: When the analysis **partitions** all lambda expressions into disjoint sets and each unknown call is associated with a unique set.
- In particular, can record the results of the analysis of [Bondorf&Jorgensen93], which can be viewed [Palsberg95] as a very conservative constraint-based analysis.
- This analysis is very useful in practice because it runs in almost-linear time.
- Have implemented as a pre-pass to closure conversion.

Removing restrictions.

- Can handle separate "compilation" by adding **extensible** types and deferring full construction of dispatch functions to "link" time.
  - Lose opportunity to optimize closure representations.
  - Can be regained within individual compilation units using rather complex scheme.
- Can remove polymorphism (where needed) as integral part of higher-order removal [Bell,Bellegarde,Hook97]
- Believe can generate first-order code without removing polymorphism, if have `typecase` operators (on types and terms).
  - Lose benefits of closure analysis

Conclusions/Ongoing Work

- Conventional algebraic types can be used to express results of closure conversion (on entire, monomorphic programs).
- Closure-converted code encodes simple but useful closure analysis.
- Standard optimizations are effective on closure-converted code.
- Will optimizations scale? Is polyvariant analysis essential?
- Would more sophisticated type systems help?
- Can we apply FORTRAN-stye optimizers to closure-converted code?