A CPS-based IR for the LLVM Backend

Johann Rudloff

Idris Developer Meeting, May 2022

May 20, 2022

CPS – Continuation Passing Style

- Every call is a tail-call
- Every function ends with a tail-call
- All function arguments are “atomic” (variable, constant)
- Functions never return (no call stack required!)
- Control flow is made explicit

Benefits of CPS

- No call stack $\rightarrow$ trivial scan for GC roots
- Full program state (current continuation + arguments) can be easily captured/suspended at any moment
  - Perfect foundation to implement lightweight threads
- Can (almost) directly be mapped to machine code
- Often cited: Easier optimisations?
- (Is this a valid reason?) Almost everyone else (in the functional space) does it: SML, several Schemes, GHC (kind of)

Transformation to CPS

- Things that would happen after a return, get wrapped in a "continuation" and passed as argument
- All functions are transformed to take a continuation as their first argument
  - This creates lots of closures, performance impact to be measured
- Instead of returning, invoke that continuation
- For LLVM, a "Lambda Lifting"-like step is required afterwards

- This can be described as "wrapping the program structure inside-out"
- In summary: Total brainfuck

Transformation to CPS - Implementation

`NamedCExp -> Core CPSExpr`

```
public export
data NamedCExp : Type where
     NmLocal : FC -> Name -> NamedCExp
     NmRef : FC -> Name -> NamedCExp
     NmLam : FC -> (x : Name) -> NamedCExp ->
     NmLet : FC -> (x : Name) -> NamedCExp ->
     NmApp : FC -> NamedCExp -> List NamedCEx
     NmCon : FC -> Name -> ConInfo -> (tag :
             List NamedCExp -> NamedCExp
     NmOp : {arity : _ } -> FC -> PrimFn arit
     NmExtPrim : FC -> (p : Name) -> List Nam
     NmForce : FC -> LazyReason -> NamedCExp
     NmDelay : FC -> LazyReason -> NamedCExp
     NmConCase : FC -> (sc : NamedCExp) -> Li
     NmConstCase : FC -> (sc : NamedCExp) ->
     NmPrimVal : FC -> Constant -> NamedCExp
     NmErased : FC -> NamedCExp
     NmCrash : FC -> String -> NamedCExp

public export
data NamedConAlt : Type where
     MkNConAlt : Name -> ConInfo -> (tag : Ma
                 NamedCExp -> NamedConAlt

public export
data NamedConstAlt : Type where
     MkNConstAlt : Constant -> NamedCExp -> N
```

```
public export
data Atom : Type where
  KLocal : FC -> Name -> Atom
  KRef : FC -> Name -> Atom
  KPrimVal : FC -> Constant -> Atom
  KErased : FC -> Atom

mutual
  public export
  data CPSExpr : Type where
    KJump : FC -> (f : Atom) -> (args : List Atom) ->
    KCon : FC -> (tag : Maybe Int) -> List Atom -> Nam
    KExtPrim : FC -> Name -> List Atom -> Name -> CPSE
    KFix : FC -> (args : List Name) -> (body : CPSExpr
    KOp : {arity : _} -> FC -> PrimFn arity -> Vect ar
    KConCase : FC -> (sc : Atom) -> List CPSConAlt ->
    KConstCase : FC -> (sc : Atom) -> List CPSConstAlt
    KCrash : FC -> String -> CPSExpr

  public export
  data CPSConAlt : Type where
    MkKConAlt : Name -> ConInfo -> Maybe Int -> List N

  public export
  data CPSConstAlt : Type where
    MkKConstAlt : Constant -> CPSExpr -> CPSConstAlt
```

- Implemented in a backend-agnostic way
- Potentially beneficial for other backends, esp. JavaScript

Memory Allocation - The Easy Part

- Functions contain no loops
- Max required memory can be statically inferred for all[1] functions
- Heap check at function entry, if space is not enough, jump to GC
  - GC is invoked with current function and all its arguments
  - Since functions never return, there is no (call-)stack to be scanned for GC roots
  - GC then "restarts" the function
- When enough heap available: simple "bump allocation"

[1] with exceptions, see next slide

Memory Allocation - The Challenging Part

- Non-trivial programs contain allocations of statically unknown size
  - String primitives: Str{Append,Cons,Reverse,Substr,Tail}
  - (Big) Integer arithmetic
  - Buffer, IOArray
- Current solution: wrap operators which require dynamic allocations in a primitive function, perform hand-crafted heap check on entry

Current Roadmap for the CPS-based LLVM Backend

- re-implement code-generation for new IR ($\sim$ 80% done)
- adjust compiler primitives and "builtins" for new allocation mechanism ($\sim$ 90% done)
- hook up the GC (prepared with stubs)
  - should be straight-forward but surprises may lurk here
- current progress: ~~1~~ ~~5~~ ~~12~~ **20 out of 22 tests passing**[2]
- the big milestone: self-hosting
- figure out "how to FFI"

[2] custom selection from Idris2 codebase + own backend-tests

Source Code

- "rapid" an Idris2 LLVM Backend - "cps" branch (active)
  https://git.sr.ht/~cypheon/rapid/tree/cps
- Idris2 CPS Transform with a dummy JS backend (bit dated)
  https://git.sr.ht/~cypheon/idris2-cps

References

- Code & Co.: Compiling With CPS
  https://jozefg.bitbucket.io/posts/2015-04-30-cps.html
- Jared Tobin: Transforming to CPS
  https://jtobin.io/transforming-to-cps
- Matt Might: How to compile with continuations
  https://matt.might.net/articles/cps-conversion/
- Appel, A. W., and Jim, T. 1989. "Continuation-passing, closure-passing style," in Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89, Austin, Texas, United States: ACM Press, pp. 293–302.
  https://dl.acm.org/doi/abs/10.1145/75277.75303
- Appel, A. W. 1992. Compiling with continuations; Cambridge University Press.