CPS Soup

A functional intermediate language

10 May 2023 – Spritely

Andy Wingo

Igalia, S.L.

CPS Soup

Compiler: Front-end to Middle-end to Back-end

Middle-end spans gap between high-level source code (AST) and low-level machine code

Programs in middle-end expressed in intermediate language

CPS Soup is the language of Guile’s middle-end

How to lower?

High-level:

```
(+ 1 (if x 42 69))
```

Low-level:

```
  cmp $x, #f
  je L1
  mov $t, 42
  j L2
L1:
  mov $t, 69
L2:
  addi $t, 1
```

How to get from here to there?

1970s

Control-flow graph (CFG)

```
graph := array<block>
block := tuple<preds, succs, insts>
inst  := const C
       | z = add x, y
       ...
```

Assignment, not definition

1980s

Static single assignment (SSA) CFG

```
graph := array<block>
block := tuple<preds, succs, phis, insts>
phi   := z := φ(x, y, ...)
inst  := const C
       | z := add x, y
       ...
```

In `v2:=φ(v0,v1)`, `v2` is

- `v0` if coming from first predecessor
- `v1` from second predecessor

Phony function

2003: MLton

Refinement: phi variables are basic block args

```
graph := array<block>
block := tuple<preds, succs, args, insts>
```

Inputs of phis implicitly computed from preds

```
BB0(a0): if a0 then BB1() else BB2()
BB1(): v0 := const 42; BB3(v0)
BB2(): v1 := const 69; BB3(v1)
BB3(v2): v3 := addi v2, 1; return v3
```

Scope and dominators

```
BB0(a0): if a0 then BB1() else BB2()
BB1(): v0 := const 42; BB3(v0)
BB2(): v1 := const 69; BB3(v1)
BB3(v2): v3 := addi v2, 1; return v3
```

What vars are “in scope” at BB3? `a0` and `v2`.

Not `v1` or `v2`; not all paths from BB0 to BB3 define `v1`.

`a0` always defined: its definition *dominates* all uses.

BB0 dominates BB3: All paths to BB3 go through BB0.

Refinement: Control tail

Often nice to know how a block ends (e.g. to compute phi input vars)

```
graph := array<block>
block := tuple<preds, succs, args, insts,
                    control>
control := if v then L1 else L2
         | L(v, ...)
         | switch(v, L1, L2, ...)
         | return v
```

Refinement: DRY

Block successors directly computable from control

Predecessors graph is inverse of successors graph

```
graph := array<block>
block := tuple<args, insts, control>
```

Can we simplify further?

Basic blocks are annoying

Ceremony about managing insts; array or doubly-linked list?

Nonuniformity: “local” vs “global” transformations

Optimizations transform graph A to graph B; mutability complicates this task

- Desire to keep A in mind while making B
- Bugs because of spooky action at a distance

Basic blocks, phi vars redundant

Blocks: label with args sufficient; “containing” multiple instructions is superfluous

Unify the two ways of naming values: every var is a phi

```
graph := array<block>
block := tuple<args, inst>
inst  := L(expr)
       | if v then L1() else L2()
       ...
expr  := const C
       | add x, y
       ...
```

Arrays annoying

Array of blocks implicitly associates a label with each block

Optimizations add and remove blocks; annoying to have dead array entries

Keep labels as small integers, but use a map instead of an array

```
graph := map<label, block>
```

This is CPS soup

```
graph := map<label, cont>
cont  := tuple<args, term>
term  := continue to L
           with values from expr
       | if v then L1() else L2()
       ...
expr  := const C
       | add x, y
       ...
```

SSA is CPS

No explicit scope tree: implicit property of control flow

CPS soup in Guile

Compilation unit is intmap of label to cont

```
cont := $kargs names vars term
      | ...
term := $continue k src expr
      | ...
expr := $const C
      | $primcall 'add #f (a b)
      | ...
```

Conventionally, entry point is lowest-numbered label

CPS soup

```
term := $continue k src expr
      | $branch kf kt src op param args
      | $switch kf kt* src arg
      | $prompt k kh src escape? tag
      | $throw src op param args
```

Expressions can have effects, produce values

```
expr := $const val
      | $primcall name param args
      | $values args
      | $call proc args
      | ...
```

Kinds of continuations

Guile functions untyped, can multiple return values

Error if too few values, possibly truncate too many values, possibly cons as rest arg...

Calling convention: contract between val producer & consumer

- both on call and return side

Continuation of `$call` unlike that of `$const`

The conts

```
cont := $kfun src meta self ktail kentry
      | $kclause arity kbody kalternate
      | $kargs names syms term
      | $kreceive arity kbody
      | $ktail
```

$kclause, $kreceive very similar

Continue to $ktail: return

$call and return (and $throw, $prompt) exit first-order flow graph

High and low

CPS bridges AST (Tree-IL) and target code

High-level: vars in outer functions in scope

Closure conversion between high and low

Low-level: Explicit closure representations; access free vars through closure

Optimizations at all levels

Optimizations before and after lowering

Some exprs only present in one level

Some high-level optimizations can merge functions (higher-order to first-order)

Practicalities

Intmap, intset: Clojure-style persistent functional data structures

Program: `intmap<label,cont>`

Optimization: `program→program`

Identify functions:
`(program,label)→intset<label>`

Edges: `intmap<label,intset<label>>`

Compute succs:
`(program,label)→edges`

Compute preds: `edges→edges`

Flow analysis

```
A[k] = meet(A[p] for p in preds[k])
         - kill[k] + gen[k]
```

Compute available values at labels:

- A: `intmap<label,intset<val>>`
- meet: `intmap-intersect<intset-intersect>`
- -, +: `intset-subtract`, `intset-union`
- kill[k]: values invalidated by cont because of side effects
- gen[k]: values defined at k

Persistent data structures FTW

- meet: intmap-intersect<intset-intersect>
- -, +: intset-subtract, intset-union

Naïve: O(nconts * nvals)

Structure-sharing: O(nconts * log(nvals))

CPS soup: strengths

Relatively uniform, orthogonal

Facilitates functional transformations and analyses, lowering mental load: “I just have to write a function from foo to bar; I can do that”

Encourages global optimizations

Some kinds of bugs prevented by construction (unintended shared mutable state)

We get the SSA optimization literature

CPS soup: weaknesses

Pointer-chasing, indirection through intmaps

Heavier than basic blocks: more control-flow edges

Names bound at continuation only; phi predecessors share a name

Over-linearizes control, relative to sea-of-nodes

Overhead of re-computation of analyses

Recap

CPS soup is SSA, distilled

Labels and vars are small integers

Programs map labels to conts

Conts are the smallest labellable unit of code

Conts can have terms that continue to other conts

Compilation simplifies and lowers programs

Wasm vs VM backend: a question for another day :)