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?
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
φ v0 if coming from first predecessor
φ v1 from second predecessor
Phony function
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)

```plaintext
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

\[
\text{graph} := \text{array}\langle \text{block} \rangle
\]

\[
\text{block} := \text{tuple}\langle \text{args, insts, control} \rangle
\]

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

\[
\begin{align*}
\text{graph} & := \text{array<block>} \\
\text{block} & := \text{tuple<args, inst>} \\
\text{inst} & := L(\text{expr}) \\
& \quad | \text{if } v \text{ then } L1() \text{ else } L2() \\
& \quad \ldots \\
\text{expr} & := \text{const } C \\
& \quad | \text{add } x, y \\
& \quad \ldots
\end{align*}
\]
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

\[
\text{graph} := \text{map}\langle\text{label}, \text{cont}\rangle
\]
\[
\text{cont} := \text{tuple}\langle\text{args}, \text{term}\rangle
\]
\[
\text{term} := \text{continue to } L
\]
\[
\quad \text{with values from } \text{expr}
\]
\[
\quad | \text{if } v \text{ then } L1() \text{ else } L2()
\]
\[
\ldots
\]
\[
\text{expr} := \text{const } C
\]
\[
\quad | \text{add } x, y
\]
\[
\ldots
\]

SSA is CPS

No explicit scope tree: implicit property of control flow
CPS soup in Guile

Compilation unit is intmap of label to cont

\[
\text{cont} := \text{kargs names vars term} \\
| \ldots \\
\text{term} := \text{continue k src expr} \\
| \ldots \\
\text{expr} := \text{const C} \\
| \text{primcall 'add #f (a b)} \\
| \ldots \\
\]

Conventionally, entry point is lowest-numbered label
CPS soup

\[
\text{term} := \$\text{continue } k \ src \ expr \\
| \$\text{branch } kf \ kt \ src \ op \ param \ args \\
| \$\text{switch } kf \ kt* \ src \ arg \\
| \$\text{prompt } k \ kh \ src \ escape? \ tag \\
| \$\text{throw } src \ op \ param \ args
\]

Expressions can have effects, produce values

\[
\text{expr} := \$\text{const } val \\
| \$\text{primcall } name \ param \ args \\
| \$\text{values } args \\
| \$\text{call } proc \ args \\
| ... \
\]
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 contents

\[
\text{cont} := \text{kfun src meta self ktail kentry} \\
| \text{kclause arity kbody kalternate} \\
| \text{kargs names syms term} \\
| \text{kreceive arity kbody} \\
| \text{ktail}
\]

$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<\text{label}, \text{cont}>

Optimization: program \rightarrow \text{program}

Identify functions:
(program, label) \rightarrow \text{intset<\text{label>}

Edges: intmap<\text{label}, \text{intset<\text{label>}}>

Compute succs:
(program, label) \rightarrow \text{edges}

Compute preds: edges \rightarrow \text{edges}
Flow analysis


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 :)