Abstract

Short presentation of Kildall's algorithm [**?**, k-unified] These notes are deliberately sparse on examples; it's hard to improve on that aspect of the original paper.

Kildall & Static Analysis

Gary Kildall (1942–1994) grew up in Seattle, attended U. Washington, taught at the US Naval Postgraduate School, and worked at Intel. He is best known for developing the `PL/M` [3] and `CP/M` [1] microprocessor programming languages.

Static Analysis is the technical term for analyzing and modifying a program *before* running it. In 1973, just after finishing his doctorate at UW, Kildall published an algorithm unifying many similar static analyses for control-flow graphs.

Preliminaries

Define a language of control-flow graphs (CFGs) with:[1]

$$\begin{aligned} \textit{Variables } v &:= x \mid y \mid \dots \\ \textit{Constants } c &:= \mathbb{Z} \mid \dots \\ \textit{Expressions } e &:= v \leftarrow e \mid e_1 + e_2 \mid e_1 - e_2 \mid \dots \end{aligned}$$

Henceforth we will use meta-variables V, C, and E to denote the set of all variables, constants, and expressions, respectively.

A *program* is given as a graph (V, E, I) where:

- V is a collection of expressions (nodes in the CFG)
- $E \subseteq V \times V$ represents directed edges between expressions in V. By convention, $E = (e_1, e_2)$ means that control flows from e_1 into e_2.
- $I \subseteq V$ is a collection of *entry points* to the program.

Lastly, define the *immediate successors* of an expression $e \in V$ as the set $\mathsf{succ}(e) = \{e' \mid (e, e') \in E\}$.

The Algorithm

Input

The algorithm requires:

- A program (V, E, I)
- A finite *pool* P of optimizing information

[1] The language as given does not allow an expression to conditionally branch to 2 or more expressions. This is an imporant feature, but orthogonal to understanding Kildall's algorithm.

- An equivalence relation $=$ on P
- A commutative, associative meet operation $\wedge : P \times P \to P$ such that the strict partial order $p_1 < p_2 \equiv p_1 \wedge p_2 = p_1 \wedge p_1 \neq p_2$ is well-founded
- A token $\star$ such that $\forall p \in P, p \wedge \star = p$
- A store $\Sigma : V \to P$ of information associated with expression nodes. Initially, $\Sigma(e) = \lambda(v).\star$. Note that we model Σ as a function, but it is very much *stateful*.
- An analysis function $f : V \times P \to P$ that propogates information through an expression
- A start function $\eta : I \to P$ of information about entry points. One possible implementation is $\lambda(i).\emptyset$.

Together, P, $\wedge$, and $\star$ form a *meet-finite* semilattice.

Intuitively, the pool is the domain of information obtained via static analysis and the analysis function refines the information associated with an expression. Running the algorithm will produce the "most refined" information for each expression in the program graph.

Algorithm

Kildall's algorithm maintains a worklist $L \in \mathcal{P}((V \times P))$ of nodes to visit, and repeatedly propogates information through the program graph. If $(e, p) \in L$, then expression e has new information p flowing into it from a predecessor expression.

We assume two basic set operations on L:

- $pop : L \to (V \times P)$ remove and return an arbitrary element of L
- $\cup : L \times L \to L$ comine two worklists, removing duplicates. (Removing duplicates is not strictly necessary, but improves performance.)

Note that there may be two elements $(e, p_1), (e, p_2)$ associated with an expression e in L.

1: $L \leftarrow \{(i, \eta(i)) \mid i \in I\}$
2: **while** $L \neq \emptyset$ **do**
3: $\quad (e, p_i) \leftarrow pop(L)$
4: $\quad p_e \leftarrow \Sigma(e)$
5: $\quad p_e^+ \leftarrow p_e \wedge p_i$
6: $\quad$ **if** $p_e^+ \neq p_e$ **then**
7: $\quad\quad \Sigma(e) \leftarrow p_e^+$
8: $\quad\quad L \leftarrow L \cup \{(e', f(e, p_e^+)) \mid e' \in \mathsf{succ}(e)\}$
9: $\quad$ **end if**
10: **end while**

Information propogates through the whole program regardless of branching structure. That is really great. No matter how many `while`-loops or `GOTO` statements in a program, Kildall's algorithm can handle it.

Properties

Theorem 1. *Kildall's algorithm runs in "polynomial time".*

Proof. Each iteration of the outer loop removes an element from L. Elements are added to L on line 8, so it suffices to show that line 8 is executed only finitely many times.

We reach line 8 only if $p_e^+ \neq p_e$, or rather, only if $p_e \wedge p_i \neq p_e$. In terms of the well-founded order $<$, this condition is $p_i < p_e$; intuitively, p_i must refine the information in Σ. Now since $p_e = \Sigma(e)$ is initialized to $\star$ and $\wedge$ is a meet operation, $p_i < p_e$ can only be true finitely often.

In particular, the running time of Kildall's algorithm is $O(|V| * \|P\|)$, where $|V|$ is the cardinality of V and $\|P\|$ is the length of the longest path from $\star$ to $\emptyset$ in the lattice defined by P and $\wedge$. □

After finishing the proof, we see the reason for the quotes around "polynomial time". Running time is polynomial with respect to the number of expressions in the program *and* the height of the optimizing lattice.

For the next theorem, let p_e^* be the result of $\Sigma(e)$ after Kildall's algorithm terminates. Also let $\mathsf{path}(e)$ be the set of all paths $e^* = e_1, \ldots, e_m, e$ from an expression $e_1 \in I$ to e and let f^* be the extension of f to paths, defined as $f^*(e^*) = f(e, f(e_m, \ldots f(e_1, \eta(e_1)) \ldots))$.

Theorem 2. $p_e^* = \bigwedge_{e^* \in \mathsf{path}(e)} f^*(e^*)$

Proof. Kildall's algorithm enumerates paths e^* of length k (henceforth e_k^*) in a breadth-first manner. The enumeration continues as long as $e_k^* > e_{k+k'}^*$ for each successive pair of paths. Conversely, once a pair $e_k^* \leq e_{k+k'}^*$ is found then $e_k^* \leq e_{k+k''}^*$ for all longer paths (of length $k + k''$). Thus Kildall's algorithm halts when $\bigwedge$ reaches a fixed-point. □

Corollary 1. *Theorem 2 holds for any implementation of* pop.

Proof. Follows from Theorem 2 and the associativity of $\wedge$. The order expressions are processed does not matter—for correctness. The order does, however, affect how quickly the algorithm converges. □

Example Passes

Constant Propogation

The goal of constant propogation is to replace variables with compile-time constants, where possible. For example, the sequence of instructions:

```
a <- 1
b <- a
```

Could easily be converted to:

```
a <- 1
b <- 1
```

and so on for more complicated expressions, if an optimizing pass associates variable references with known constants. To this end, we can implement the parameters for Kildall's algorithm:

- The pool P is all sets of all pairs of variables and constants: $\mathcal{P}((V \times C))$.
- The meet operation $\wedge$ is set intersection; the empty set means we have no information about constants in an expression.
- The optimizing function f is such that $(v, c) \in f(e, p)$ if and only if:
  - $(v, c) \in p$ and e is not an assignment $v < -e'$
  - e is an assignment $v < -c$

After computing Σ, a final pass through the program can substitute constants.

Common-Subexpression Elimination

CSE replaces computations with variable references when the computation has previously been stored in a variable. As a quick example:

```
a <- 1234 * 5678
b <- 1234 * 5678
```

Only needs to perform a single multiplication. Other pure computations can also be memoized.

In this analysis:

- The pool is all sets of equivalence classes of expressions, $\mathcal{P}(\mathcal{P}(E))$
- The meet operation is pointwise intersection of equivalence classes:

$$p_1 \wedge p_2 = \{p_1' \cap p_2' \mid p_1' \in p_1 \wedge p_2' \in p_2\}$$

- The optimizing function puts sub-expressions into equivalence classes and tries to replace expressions with variable references.
  - First, traverse the AST of the node e and replace known sub-expressions with variable references.
  - Create a new equivalence class for e, add all expressions with sub-expressions equivalent to a sub-expression in e
  - If e is an assignment $a < -e'$, remove all references to a in the pool and make copies of all expressions containing e', substituting a for e' in the copy.

Register Optimization

Register optimization seeks to minimize the number of variables stored in registers at a program node. For example the following expression needs space to store 4 computations:

```
a <- b + c
```

One for b, one for c, one for $b + c$, and one for a. If b and c are not used in subsequent expressions, then their registers may be recycled.

Kildall's algorithm can determine the number of occupied registers at each node by reversing edges in the program graph:

- The pool is $\mathcal{P}(V \cup E)$, because variables and expressions require register space.
- The meet is set union; in the worst case, we need registers for every variable in the program.
- The optimizing function records the variables referenced in a node:
  - If e is an assignment to v, remove v from the current pool.
  - Add every *sub-expression* in the AST of e to the pool. For instance, $a + b$ creates a pool $\{a, b, a + b\}$.

Modern Applications

CompCert

The CompCert compiler [5, 6][2] uses Kildall's algorithm in 8 of its 20[3] major passes.

The optimized passes are:

Pass Title	*Explanation*
Allocation	Register allocation
ConstProp	Constant Propogation
CSE	Common-Subexpression Elimination
DeadCode	Dead-Code Elimination
Linearize	Linearize a control-flow graph
Liveness	Liveness analysis for registers
RegAlloc	Register Allocation
Splitting	Split registers' live range

The End of Optimizing Compilers?

When preparing these notes, I downloaded a few compilers and grepped their sources for the string "kildall" (case insensitive).

[2] Version 2.6, as of `2015-01-24`.
[3] See the "Compiler Passes" section of: `http://compcert.inria.fr/doc/index.html`

`gcc`	`ghc`	`racket`
`clang`	`ocaml`	`CompCert`
`llvm`	`ceylon`	
`javac (jdk 8)`	`lua jit`	

Only CompCert returned matches. Maybe because there are faster algorithms today.

After a little more digging, I found the slides for a talk titled *The Death of Optimizing Compilers* [2], which claims that optimizing compilers are becoming less useful today.

- Improving hardware gives more power *and* throughput, so we tend to see very small patches of hot code and lots of little-used code.
- The largest bottlenecks are due to bad algorithms. Optimizing compilers do not convert bad algorithms into good ones (i.e. convert bubble sort into merge sort).

Makes sense to me. The benefits of constant propogation seem tiny in the grand scheme of writing a better algorithm, and experiments like *Stabilizer* [4] give evidence that the best optimization technology has hit a ceiling.

References

[1] CP/M Plus programmers guide. 1983. http://www.cpm.z80.de/manuals/cpm3-pgr.pdf.

[2] Daniel A. Bernstein. The death of optimizing compilers. 2015. https://cr.yp.to/talks/2015.04.16/slides-djb-20150416-a4.pdf.

[3] Intel Corporation. PL/M-80 programming manual. 1980. http://bitsavers.trailing-edge.com/pdf/intel/PLM/9800268B_PLM-80_Programming_Manual_Jan80.pdf.

[4] Charlie Curtsinger and Emery Berger. Stabilizer: Statistically sound performance evaluation. In *ASPLOS*, 2013. http://plasma.cs.umass.edu/emery/stabilizer.html.

[5] Xavier Leroy. The CompCert verified C compiler commented coq development. 2015. http://compcert.inria.fr/doc/index.html Last updated 2015-06-12.

[6] Xavier Leroy. The CompCert verified C compiler documentation and user's manual. 2015. http://compcert.inria.fr/man/manual.pdf Last updated 2015-06-11.