More type canonicalisation experiments

Andreas Rossberg

Recap

GC MVP currently uses equi-recursive types

Type canonicalization recently implemented in Binaryen

Thomas presented first numbers (thanks for that!)

- not promising, about 150ms for J2CL sample module

Aske observed massive slowdown in Binaryen since addition

- scary, about 12s for round-trip on dart2wasm module

Another Prototype
Type Canonicalizer

Hacked into reference interpreter

Canonicalizes entire type section of every module

Stores types in global type cache (like an engine would)

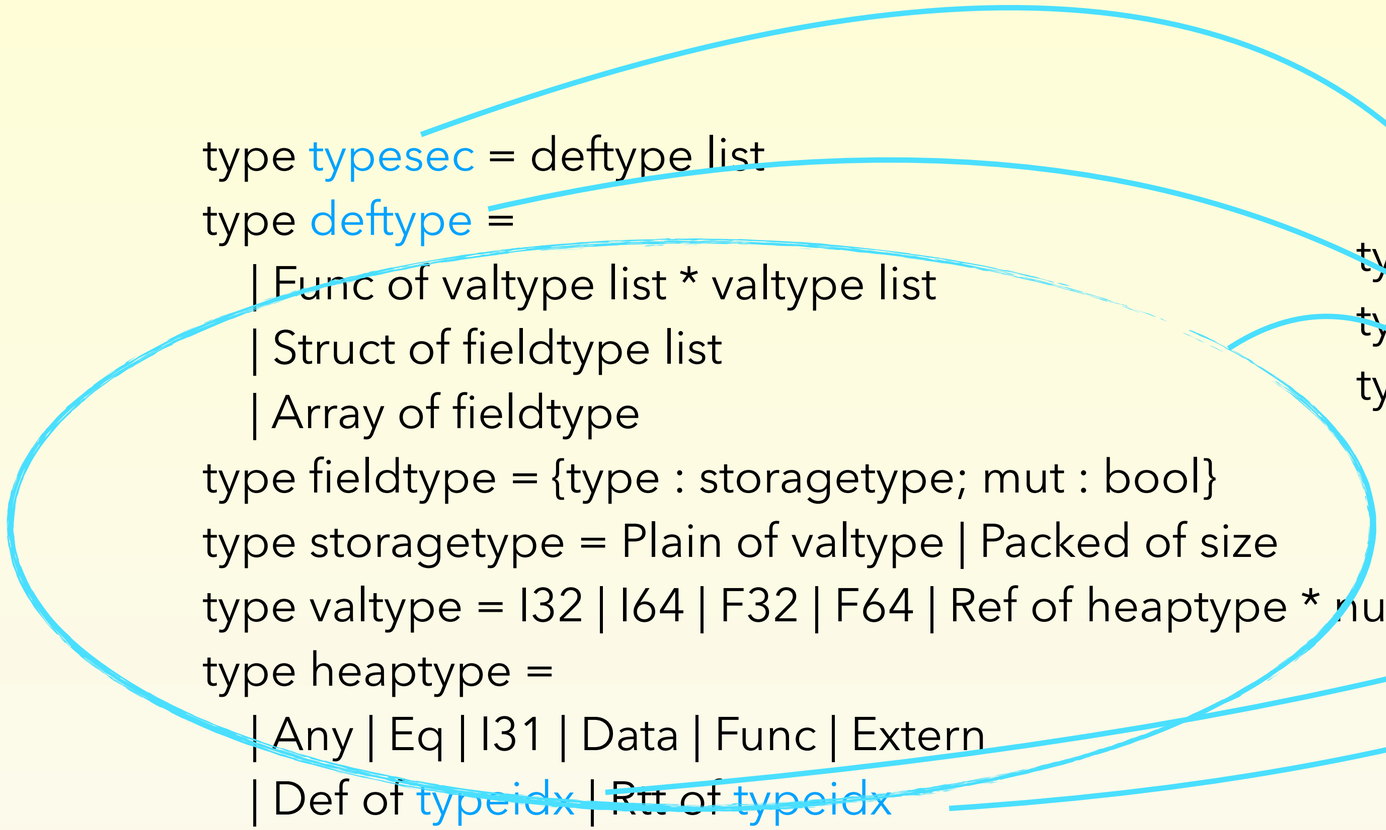
Two variants explored: whole-module vs incremental

NB: Not using the type cache for anything

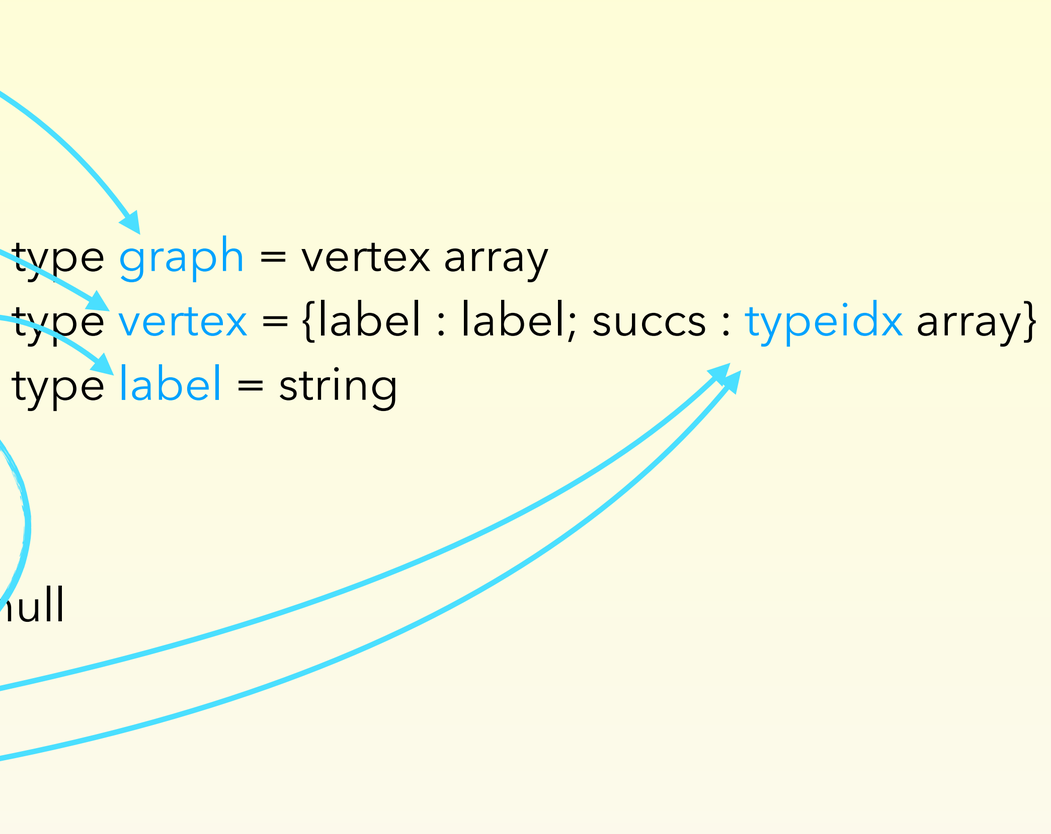
Whole-Module Canonicalization

1. Construct graph from type section
2. Run minimization on graph
3. Compute strongly-connected components on result
4. For each SCC, look up one vertex in cache
5. If not found, add SCC

Graph Construction



```
type typesec = deftype list
type deftype =
  | Func of valtype list * valtype list
  | Struct of fieldtype list
  | Array of fieldtype
type fieldtype = {type : storagetype; mut : bool}
type storagetype = Plain of valtype | Packed of size
type valtype = I32 | I64 | F32 | F64 | Ref of heaptype * null
type heaptype =
  | Any | Eq | I31 | Data | Func | Extern
  | Def of typeidx | Rtt of typeidx
```



```
type graph = vertex array
type vertex = {label : label; succs : typeidx array}
type label = string
```

NB: label implies arity

Minimization

Equivalent to DFA minimization

- alphabet A corresponds to maximum of arity of all vertices (statically unbounded)

- initial partition by vertex label, not final states

Pick a suitable algorithm

- standard Hopcroft [1970] is $O(|A|\ |V|\ \log\ |V|)$, best for total DFA

- but we have a partial DFA, with many edges missing (labels with less than max-arity)

- better algorithms exist, e.g. Valmari/Lehtinen [2008] is $O(|E|\ \log\ |V|)$

- that is, runs sub-linear in size of type section (initialization of course is still linear)

SCC Computation

Each SCC corresponds to one recursive group

Find to reduce follow-up work

Using standard algorithm from Tarjan [1972]

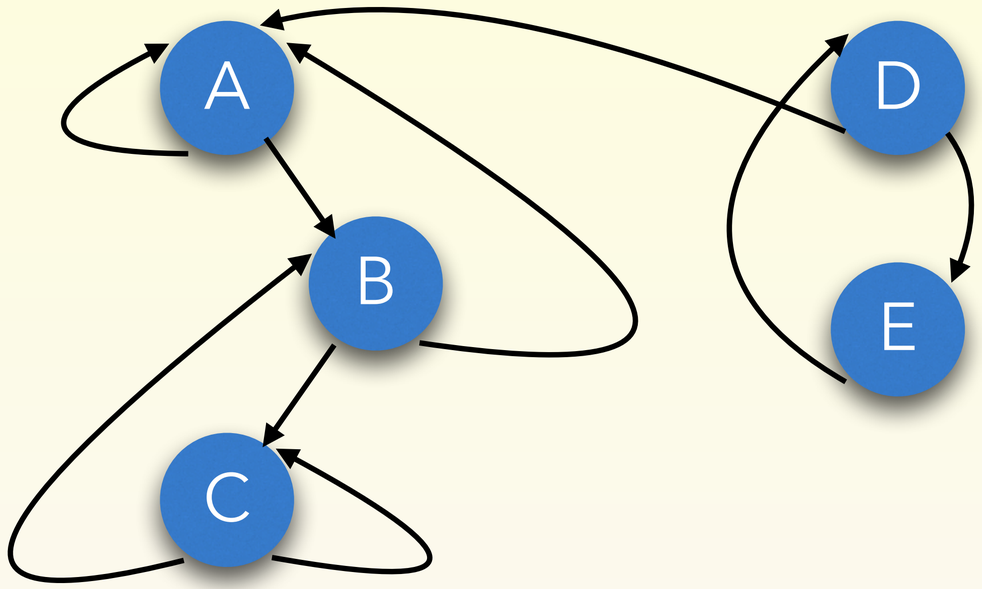
Implicitly produces topological sort

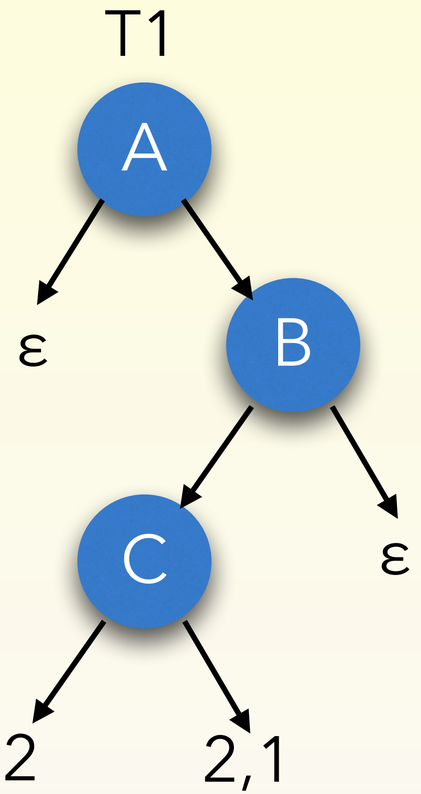
Type Cache

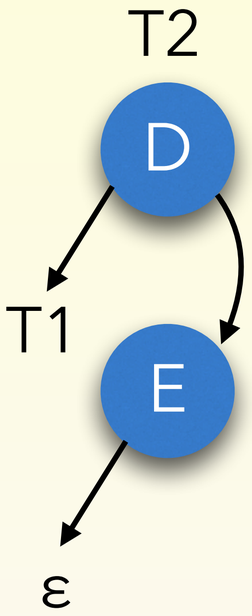
```
module Repo {
   type typeid = int
   type compid = int

   type typeinfo = {comp : compid; vertidx : int}
   type compinfo = {verts : array(vertex)}

   id_table : arraytbl(typeid, typeinfo)
   comp_table : arraytbl(compid, compinfo)

   type key =  Node(label, array(typeid | key))  |  Path(list(int))
   key_table : hashtbl(key, typeid)

   add_graph(graph) : array(typeid)
}
```

Tree Keys

type key = Node(label, array(typeid | key)) | Path(list(int))







Cache lookup

Keying assumes that SCCs are handled in topological order

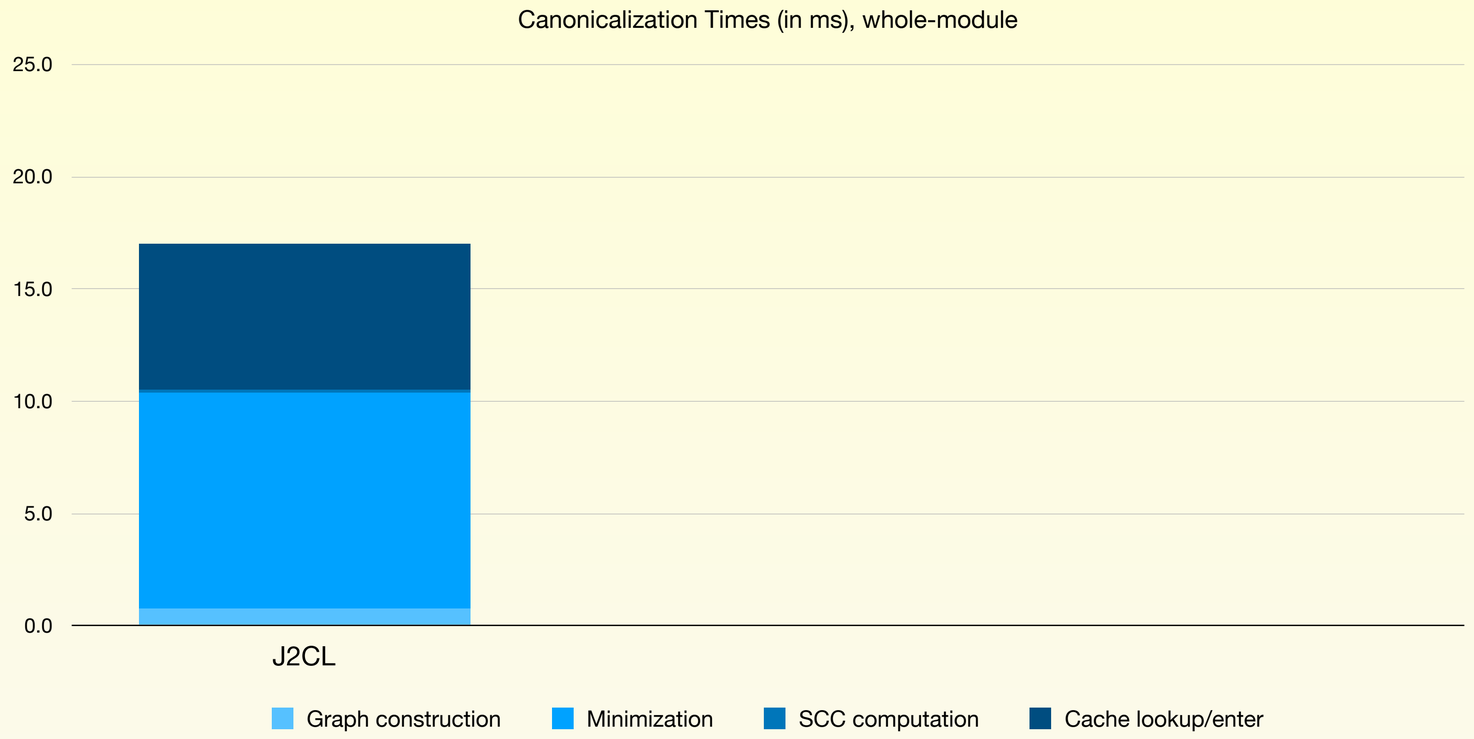
For each SCC, it is sufficient to look up one vertex

If found, parallel-traversal of external and cached SCC finds all other ids of same SCC in $O(|V_{scc}|)$

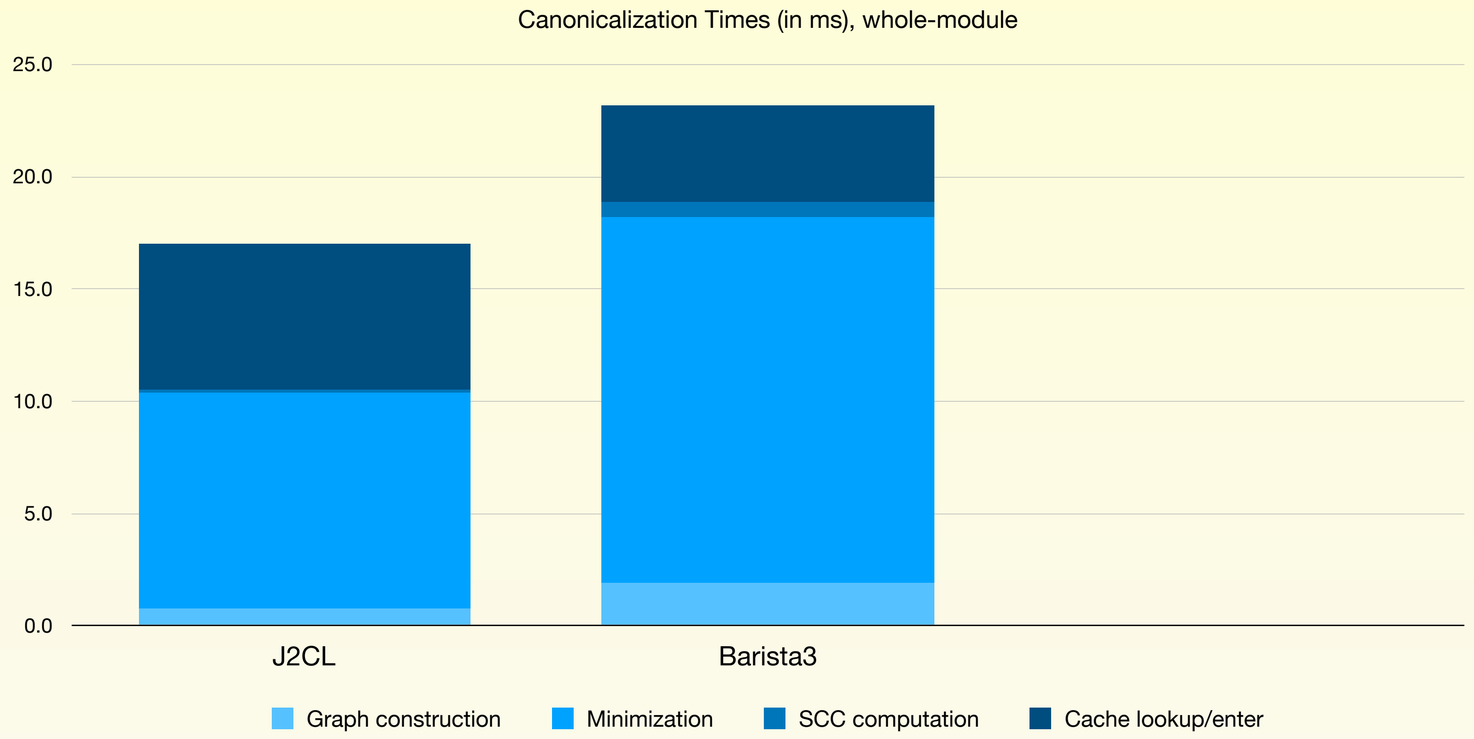
If not found, need to compute key for each vertex, which is $O(|V_{scc}|^2)$

- there is a way to avoid this entirely if all SCCs are put into a canonical vertex order, which can be computed in $O(|V_{scc}| \log |V_{scc}|)$, but haven't implemented that yet

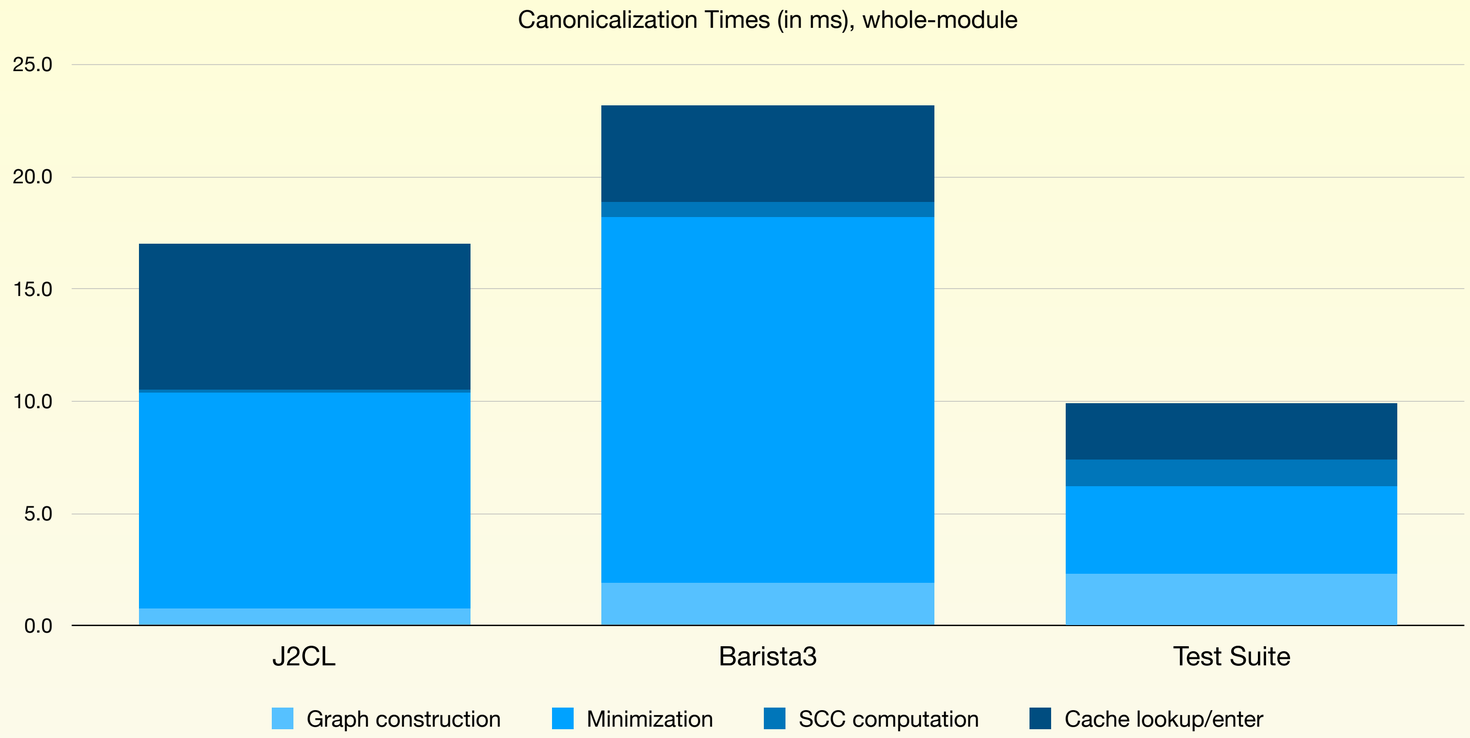
Measurements



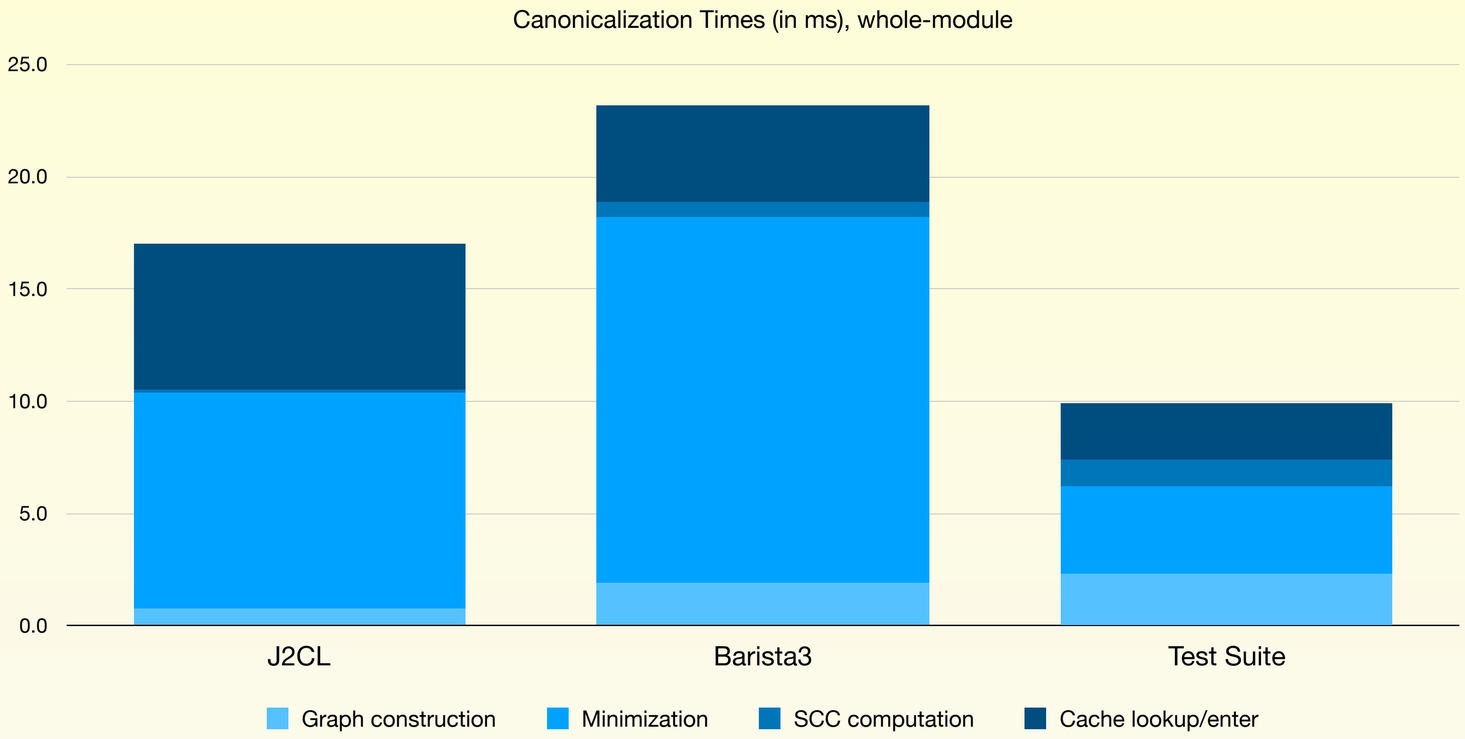
Measurements



Measurements

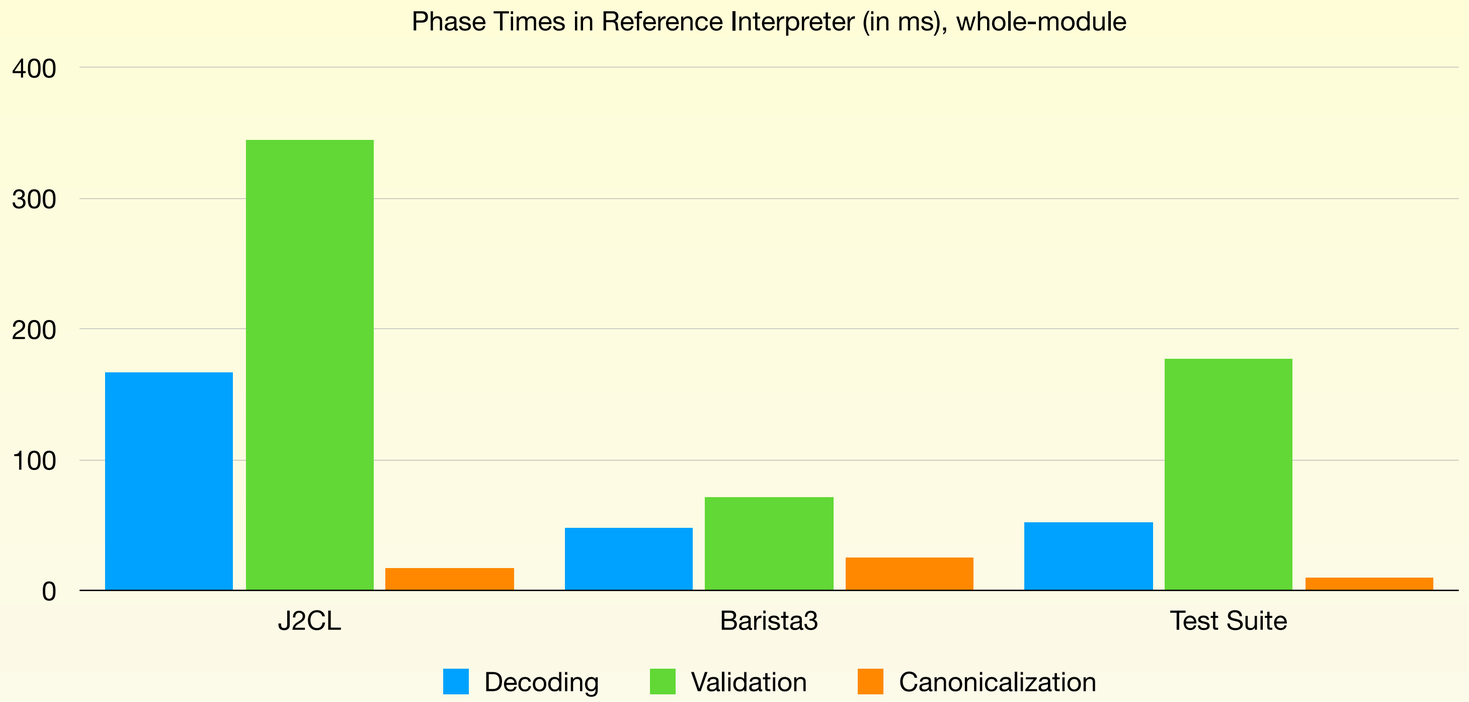


Measurements



	Types initial	Types minimized	Largest SCC	Graph construction	Minimization	SCC computation	Cache lookup/enter	GCs (minor)
J2CL	2 267	1 416	95	0.8	9.6	0.1	6.5	10
Barista3	6 970	5 566	4	1.9	16.3	0.7	4.3	13
Test Suite	1 907	205	5	2.3	3.9	1.2	2.5	0

Relative Timings



NB: For comparability, modified validator to avoid quadratic lookups

Limitations

This is all you need in a tool like Binaryen

Whole-module minimisation assumes that modules are closed

That is, all vertices of an SCC are defined in the same module

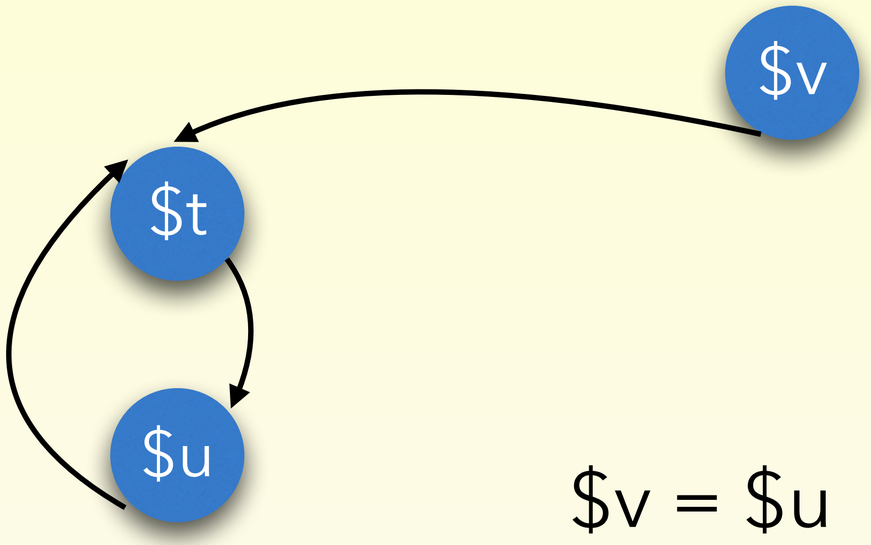
Not true anymore in an engine with type imports!

A type may be an unrolling of an import

Example

```
;; “A”
(module
  (type $t (struct (field (ref $u))))
  (type $u (struct (field (ref $t))))
  (export “t” (type $t))
)

;; “B”
(module
  (import “A” “t” (type $t))
  (type $v (struct (field (ref $t))))
)
```

Example



$v = $u

If both are in a single module, minimization will unify them

But if in separate modules, minimization happens separately

But: unrolling can only happen with directly adjacent SCCs!

Incremental Canonicalization

1. Construct graph from type section
2. Compute strongly-connected components on graph (do not minimize yet)
3. For each SCC in topological order:
    a. Compute set of adjacent vertices in cache (edges external to current SCC) that are themselves part of an SCC
    b. Minimize SCC along with corresponding adjacent set of SCC's
    c. If result has no more vertices than the adjacent set, then the SCC is known
    d. Otherwise, add to cache as before

Plus a few clever shortcuts to avoid minimization when not necessary

[cf. Laurent Mauborgne. *An Incremental Unique Representation for Regular Trees*. Nordic Journal of Computing, 7(4), 2000]

Computing the adjacent set

This is easy:

Lookup each typeid referenced by SCC

If that type's SCC is cyclic, add to set

- all SCC's of size > 1 are cyclic

- for size 1 use extra bit (or index sentinel)

Can filter further by maintaining, for each SCC,
a set of outgoing edges (label, pos, typeid)

Shortcuts before Minimization

If the graph is of size 1 and has no internal edges nor adjacent SCCs:

- type is not recursive or only with itself, look it up right away
  (this is the most common case by far)

Otherwise, maybe the SCC is already minimal, so first try to compute one of its keys:

- if that can be found in cache, SCC was minimal and already known

Otherwise, if size of new SCC is larger than log of size of adjacent set:

- naive comparison with adjacent SCC's is cheaper than minimisation

Only if all that failed, we need to proceed to minimize combined SCC + adjacents

Possible Cases after Minimization

1. SCC is unrolling of one of the adjacent SCCs

   - detected when minimized graph has same size as adjacent sets

   - then in each partition that has a vertex from the new SCC, there is exactly one representative of a cached vertex

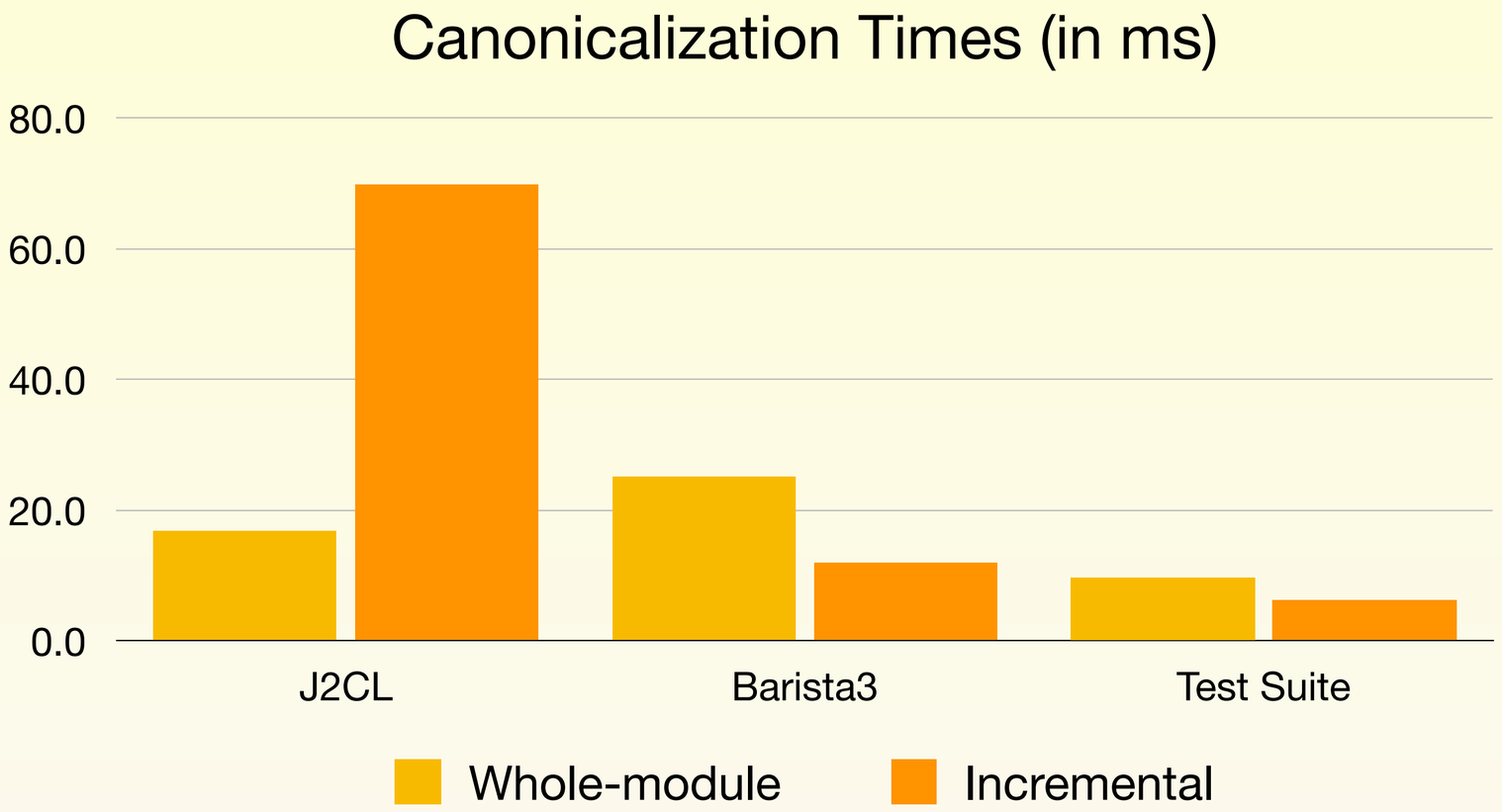
2. SCC exists in cache, but is not an unrolling

   - then we need to compute a key for one vertex, as before

3. SCC is not in cache

   - then key lookup will fail and we need to add the SCC, as before

Measurements



Warm Cache

Hm, is this worth it?

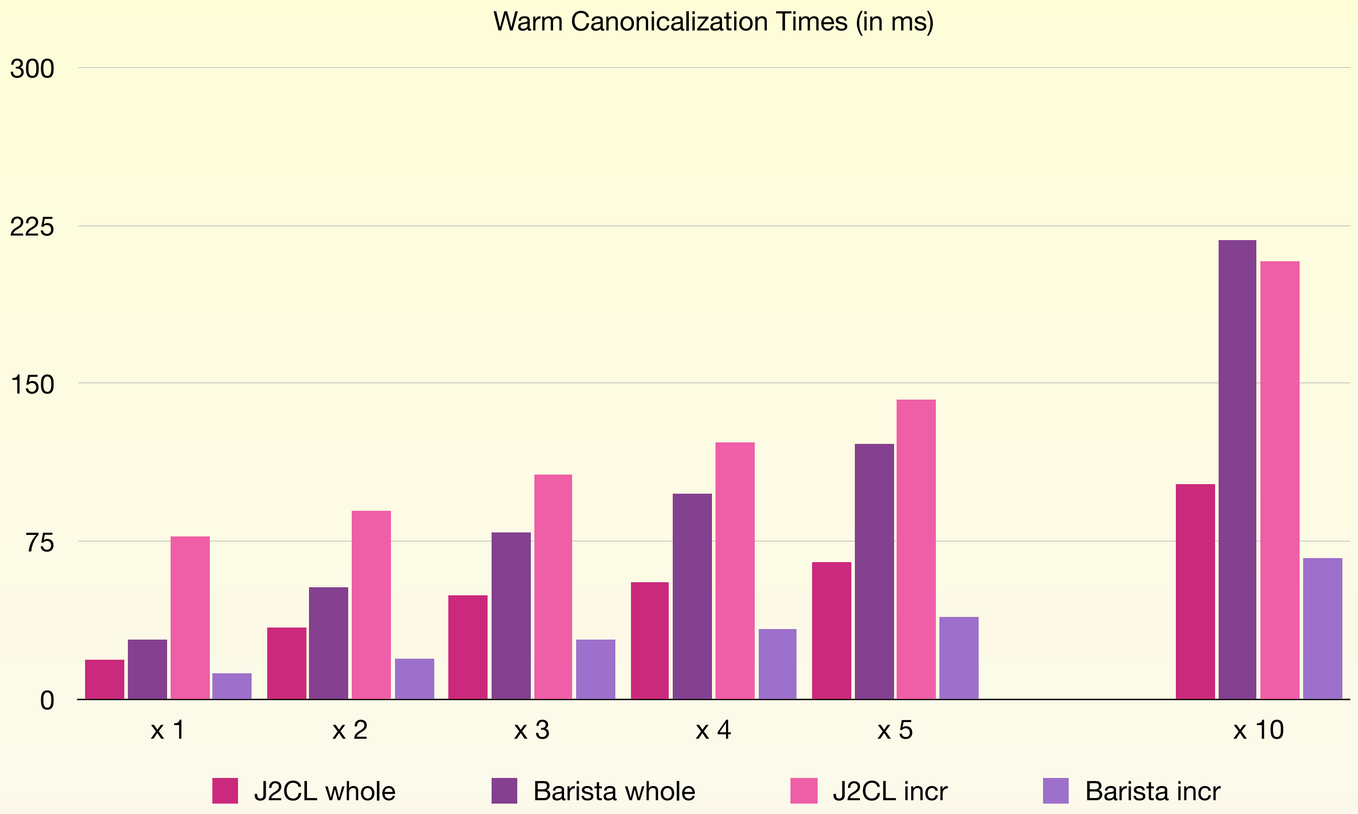
So far, we have measured cold start-up

Incremental canonicalization can benefit more from warm cache

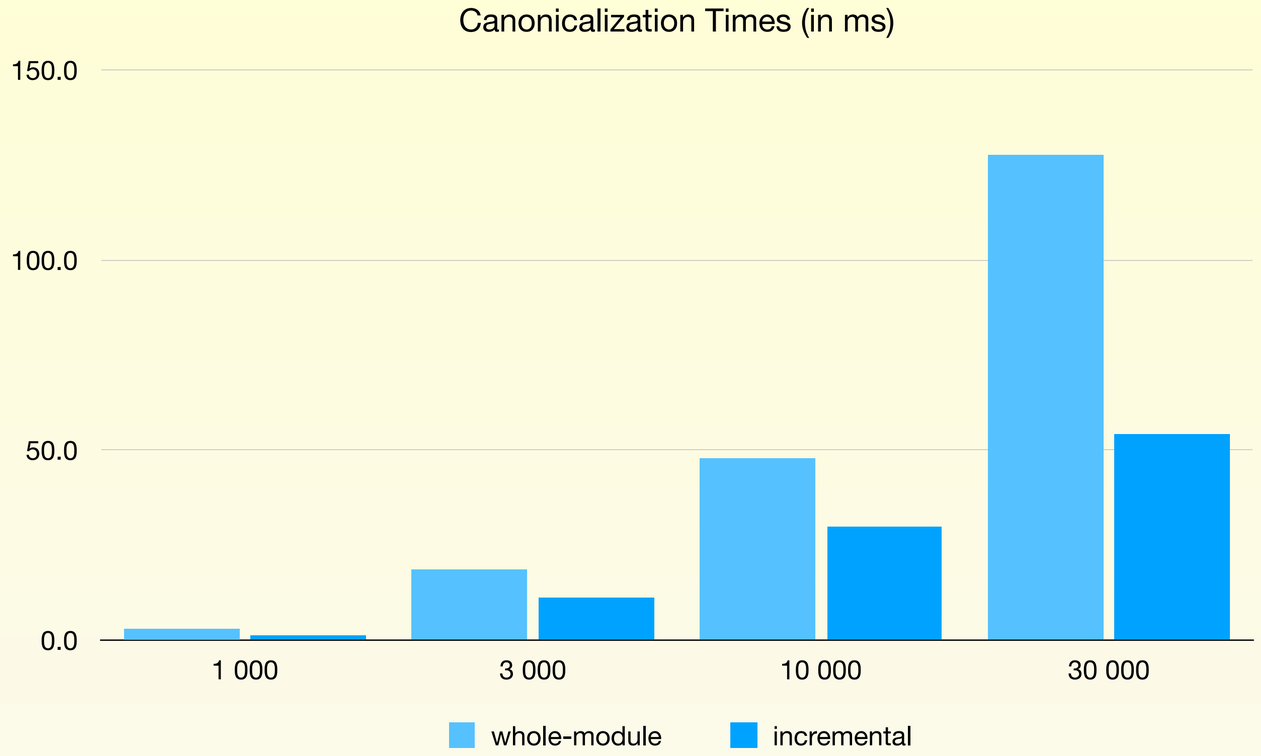
- shortcuts commonly find SCC early

- can avoid minimization altogether in many cases

Warm Cache Measurements



Fuzzing



Types	Type section (KiB)	Whole-module	Incremental
1 000	10.8	3.0	1.1
3 000	34.0	18.6	11.2
10 000	115.5	47.6	29.9
30 000	373.5	127.6	54.2

Some Thoughts

Trade-off depends on input

Could use heuristics to choose

Hybrid approach would also be possible

- biggest cost in incremental approach is minimization including adjacent SCCs

- when combined with whole-module, this is only needed for types referring to imports

Further Remarks

This is a prototype in OCaml

Essentially, all algorithms over arrays of (records of) ints

Worst-case scenario for OCaml

- arrays are bounds-checking, all records are boxed, all ints tagged

- allocations and GC happen throughout and are included in timings
  (e.g., 589 minor collections during incremental canonicalization of J2CL)

Using linked lists, list reversals, and higher-order functions in some core loops

Room for more algorithmic improvements (in particular, quadratic key computation)

Conjecture: an implementation closer-to-the-metal could shave off another 3x or more

On the other hand, it doesn't need to synchronize cache access across threads

Implementation is not trivial

Potential Parallelization

Handling of SCCs can be (mostly) parallel

- with incremental approach,
  particularly minimization

Don’t know if there are parallel algorithms for minimization itself

Canonicalization parallel to compilation?

What does this all mean?

We need more data!

Ideally, corpus of modules from multiple sources

- CG: please point us to other relevant modules

Still worth exploring backup options

Intend to talk about iso-recursive alternative next time

References

Robert Tarjan
Depth-first search and linear graph algorithms
SIAM Journal on Computing, 1(2), 1972

Antti Valmari, Petri Lehtinen
Efficient minimization of DFAs with partial transition functions
Symposium on Theoretical Aspects of Computer Science, 2008

Laurent Mauborgne
An Incremental Unique Representation for Regular Trees
Nordic Journal of Computing, 7(4), 2000

https://github.com/WebAssembly/gc/tree/canon/interpreter/canon