Lessons of Airac5

Jaeho Shin ⟨netj@ropas.snu.ac.kr⟩

March 31 and April 3, 2006

Weekly Show & Tell
Programming Research Laboratory,
Seoul National University
http://ropas.snu.ac.kr/talk/2006/

Contents

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

Goal

- Understand what's going on inside Airac5
- See why naïve approach fails in verifying real C programs
- Think about what we can do to make Airac5 stronger

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

Abstract memory *Mem*

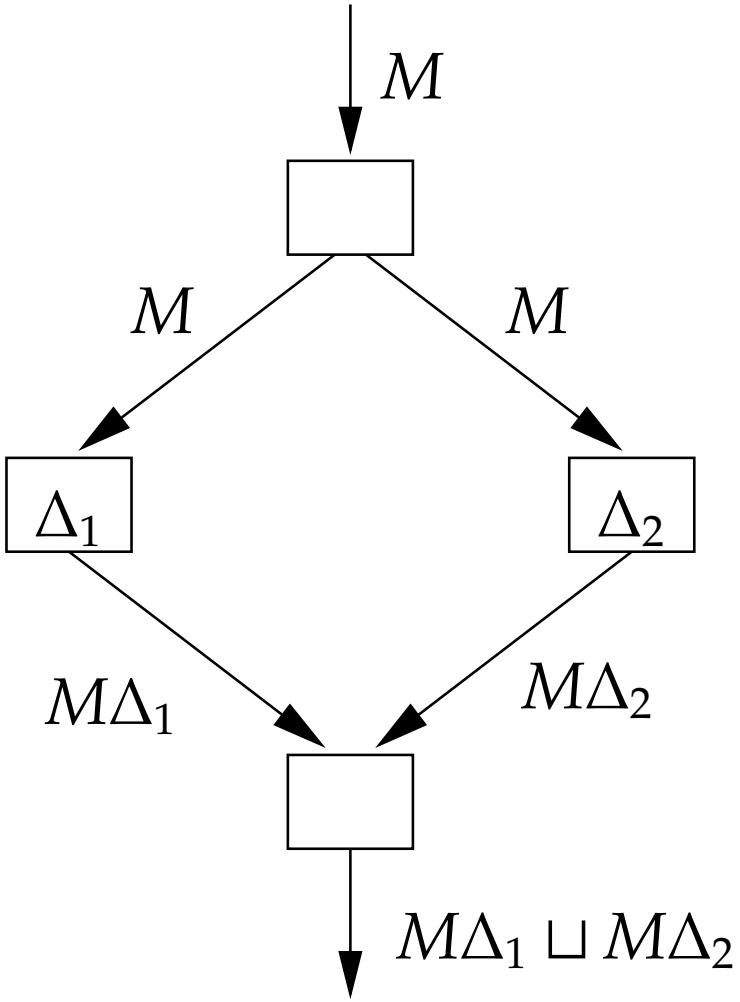
Abstract address *Addr*, abstract value *Val*

Empty memory	$\bot$	:	Mem
Lookup	$\cdot ? \cdot$	:	$Mem \to Addr \to Val$
Update	$\cdot \{ \cdot \mapsto \cdot \}$	:	$Mem \to Addr \to Val \to Mem$
Operators	$\sqcup, \triangledown, \triangle$	:	$Mem \to Mem \to Mem$

Variables all over

- Airac5 assigns each variable a unique name over the entire program.
- Treats every variable as global, i.e. no local variable.
- Every program point's abstract memory contains entries for variables all over the program.
- Huge burden for $\sqcup$, $\triangledown$ and $\triangle$

Motivation



While computing $M\Delta_1 \sqcup M\Delta_2$, why should we bother with unchanged entries from M? Can't we just look at Δ_1 and Δ_2 only?

References

Similar idea was addressed in §6.2 of


B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival.
Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter.
The Essence of Computation: Complexity, Analysis, Transformation., 2002.

Observation: idempotent operators

Our join, widening, narrowing operators are idempotent.

$$\begin{aligned} x \sqcup x &= x \\ x \triangledown x &= x \\ x \triangle x &= x \end{aligned}$$

So, we are OK to concentrate only on the changed ones.

Journaling memory *JMem*[1]

Let's keep a journal of changes $\langle m \rangle_\delta$

$$\begin{array}{lllllll} m & \in & JMem & & j & \in & Jnl \\ m & ::= & j\,M & & j & ::= & \cdot \mid \langle m \rangle_\delta \end{array}$$

where $M \in Mem$ and $\delta \subseteq Addr$.

A journaling memory m is either

- "$\cdot\, M$" with no journal, or
- "$\langle m \rangle_\delta\, M$" with a journal that means: "compared to m, entries of addresses in δ have been changed among M."

[1] Joint work with Hakjoo

Core operators on *JMem*

Operators for controlling the level of journals

- Wrap

$$\begin{aligned} wrap &: JMem \to JMem \\ wrap(j\ M) &= \langle j\ M \rangle_{\emptyset}\ M \end{aligned}$$

- Peel

$$\begin{aligned} peel &: JMem \to JMem \\ peel(\cdot\ M) &= \cdot\ M \\ peel(\langle \cdot\ M_0 \rangle_{\delta_1}\ M_1) &= \cdot\ M_1 \\ peel(\langle \langle m \rangle_{\delta_0}\ M_0 \rangle_{\delta_1}\ M_1) &= \langle m \rangle_{\delta_0 \cup \delta_1}\ M_1 \end{aligned}$$

Proxy operators on *JMem*

Corresponding operators on *Mem*

- Lookup

$$\begin{array}{rcl} \cdot?\cdot & : & JMem \to Addr \to Val \\ (\cdot\ M)?a & = & M?a \\ (\langle m \rangle_\delta\ M)?a & = & M?a \end{array}$$

- Update

$$\begin{array}{rcl} \cdot\{\cdot \mapsto \cdot\} & : & JMem \to Addr \to Val \to JMem \\ (\cdot\ M)\{a \mapsto v\} & = & \cdot\ M\{a \mapsto v\} \\ (\langle m \rangle_\delta\ M)\{a \mapsto v\} & = & \langle m \rangle_{\delta \cup \{a\}}\ M\{a \mapsto v\} \end{array}$$

(add the address to the outermost journal if it exists)

Difference between two *JMems*

$$\otimes \quad : \quad JMem \to JMem \to \wp(Addr)$$

$$\begin{array}{rcll}
(\cdot\ M_1) & \otimes & (\cdot\ M_2) & = \quad dom(M_1) \cup dom(M_2) \\
(\langle m_1 \rangle_{\delta_1}\ M_1) & \otimes & (\cdot\ M_2) & = \quad dom(M_1) \cup dom(M_2) \\
(\cdot\ M_1) & \otimes & (\langle m_2 \rangle_{\delta_2}\ M_2) & = \quad dom(M_1) \cup dom(M_2)
\end{array}$$

$$\begin{array}{rcll}
(\langle m \rangle_{\delta_1}\ M_1) & \otimes & (\langle m \rangle_{\delta_2}\ M_2) & = \quad \delta_1 \cup \delta_2 \\
(\langle m \rangle_{\delta_1}\ M_1) & \otimes & (m) & = \quad \delta_1 \\
(m) & \otimes & (\langle m \rangle_{\delta_2}\ M_2) & = \quad \delta_2
\end{array}$$

$$(m_1^{n_1}) \otimes (m_2^{n_2}) = \begin{cases} (peel(m_1)) \otimes (m_2) & \text{if } n_1 > n_2 \\ (m_1) \otimes (peel(m_2)) & \text{otherwise} \end{cases}$$

Common journal of two *JMem*s

$$\odot \quad : \quad JMem \to JMem \to Jnl$$

$$\begin{aligned}
(\cdot\ M_1) &\odot (\cdot\ M_2) &&= \cdot \\
(\langle m_1 \rangle_{\delta_1}\ M_1) &\odot (\cdot\ M_2) &&= \cdot \\
(\cdot\ M_1) &\odot (\langle m_2 \rangle_{\delta_2}\ M_2) &&= \cdot
\end{aligned}$$

$$\begin{aligned}
(\langle m \rangle_{\delta_1}\ M_1) &\odot (\langle m \rangle_{\delta_2}\ M_2) &&= \langle m \rangle_{\emptyset} \\
(\langle m \rangle_{\delta_1}\ M_1) &\odot (m) &&= \langle m \rangle_{\emptyset} \\
(m) &\odot (\langle m \rangle_{\delta_2}\ M_2) &&= \langle m \rangle_{\emptyset}
\end{aligned}$$

$$(m_1^{n_1}) \odot (m_2^{n_2}) = \begin{cases} (peel(m_1)) \odot (m_2) & \text{if } n_1 > n_2 \\ (m_1) \odot (peel(m_2)) & \text{otherwise} \end{cases}$$

Selective operators on *JMem*

We define $\star \in \{\sqcup, \triangledown, \triangle\}$, given its corresponding operator $\star_\delta$ on *Mem* that considers only the entries whose address is in δ, as:

$$\star \quad : \quad JMem \to JMem \to JMem$$

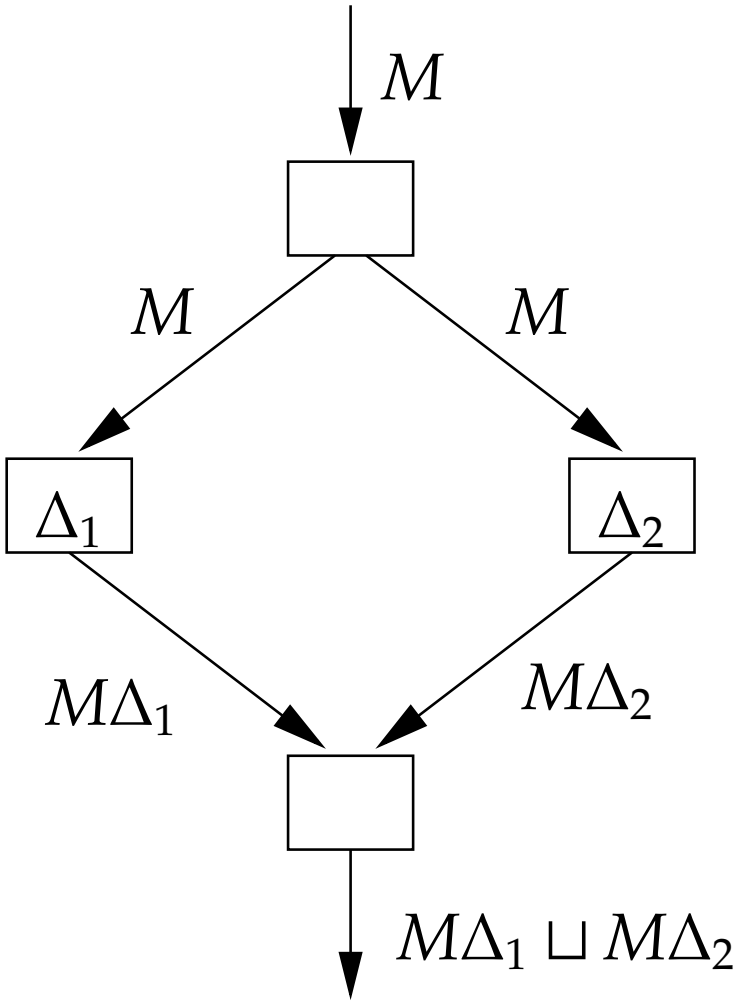
$$(j_1\ M_1) \star (j_2\ M_2) \quad = \quad j\ (M_1 \star_\delta M_2)$$

$$\text{where} \quad \delta \quad = \quad (j_1\ M_1) \otimes (j_2\ M_2)$$

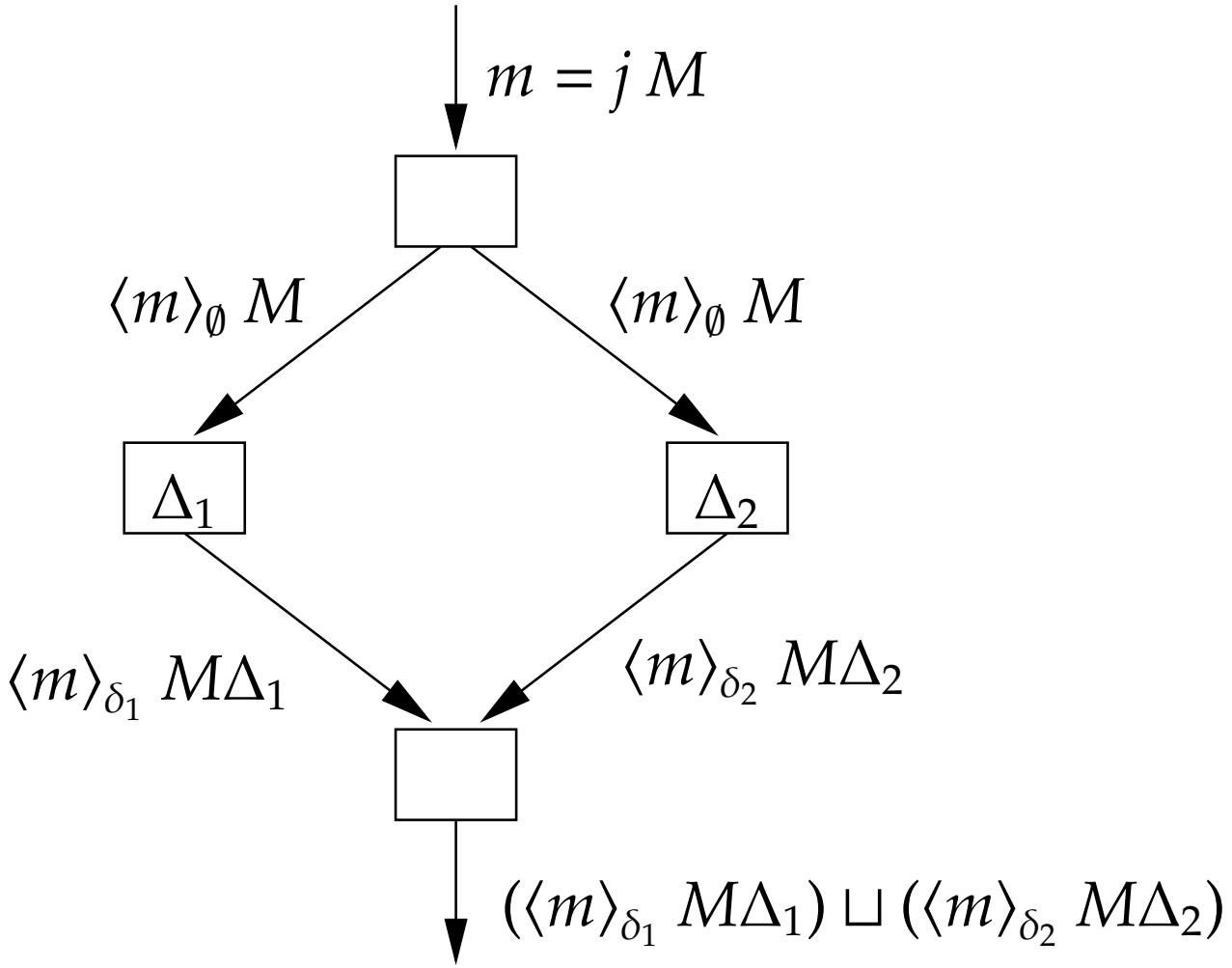
$$\text{and} \quad j \quad = \quad \begin{cases} \langle m \rangle_{\delta' \cup \delta} & \text{if } \langle m \rangle_{\delta'} = (j_1\ M_1) \odot (j_2\ M_2) \\ \cdot & \text{otherwise} \end{cases}$$

- Localizing operators
  - Motivation
  - Journaling memory
  - Selective operators
  - Experiment results

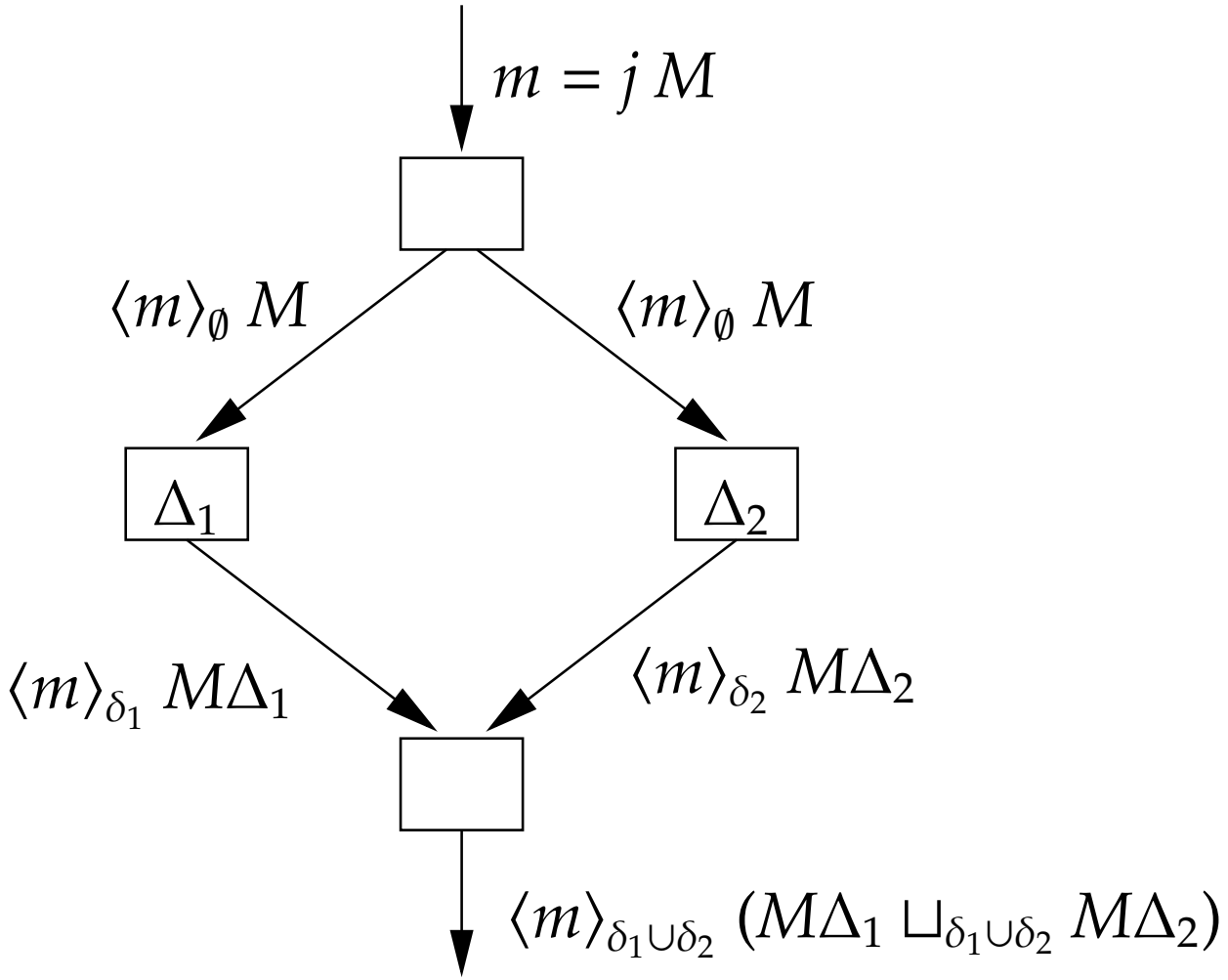
Example: the branch (without selective operators)



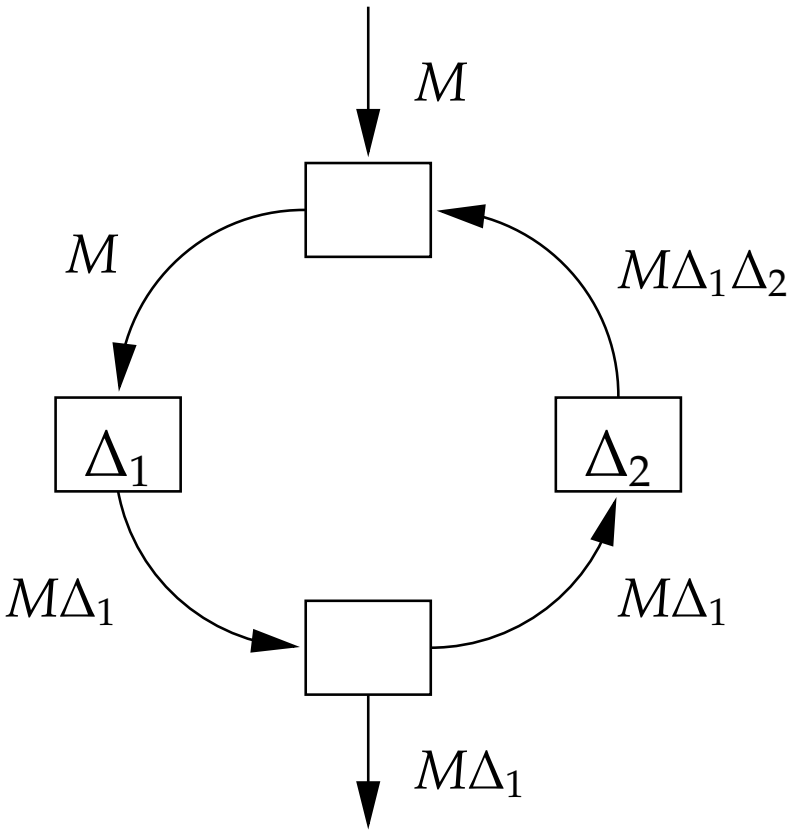
Example: the branch



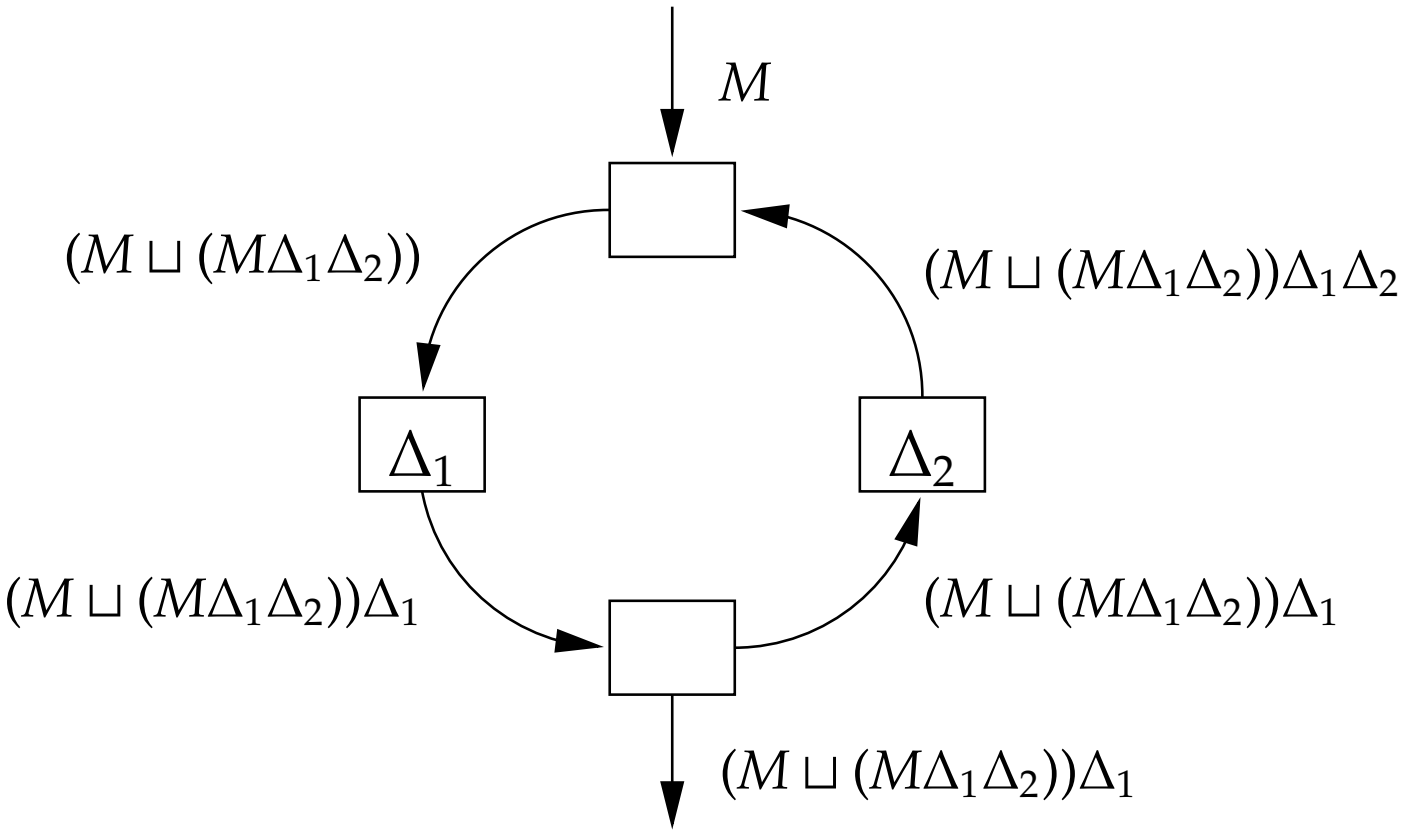
Example: the branch



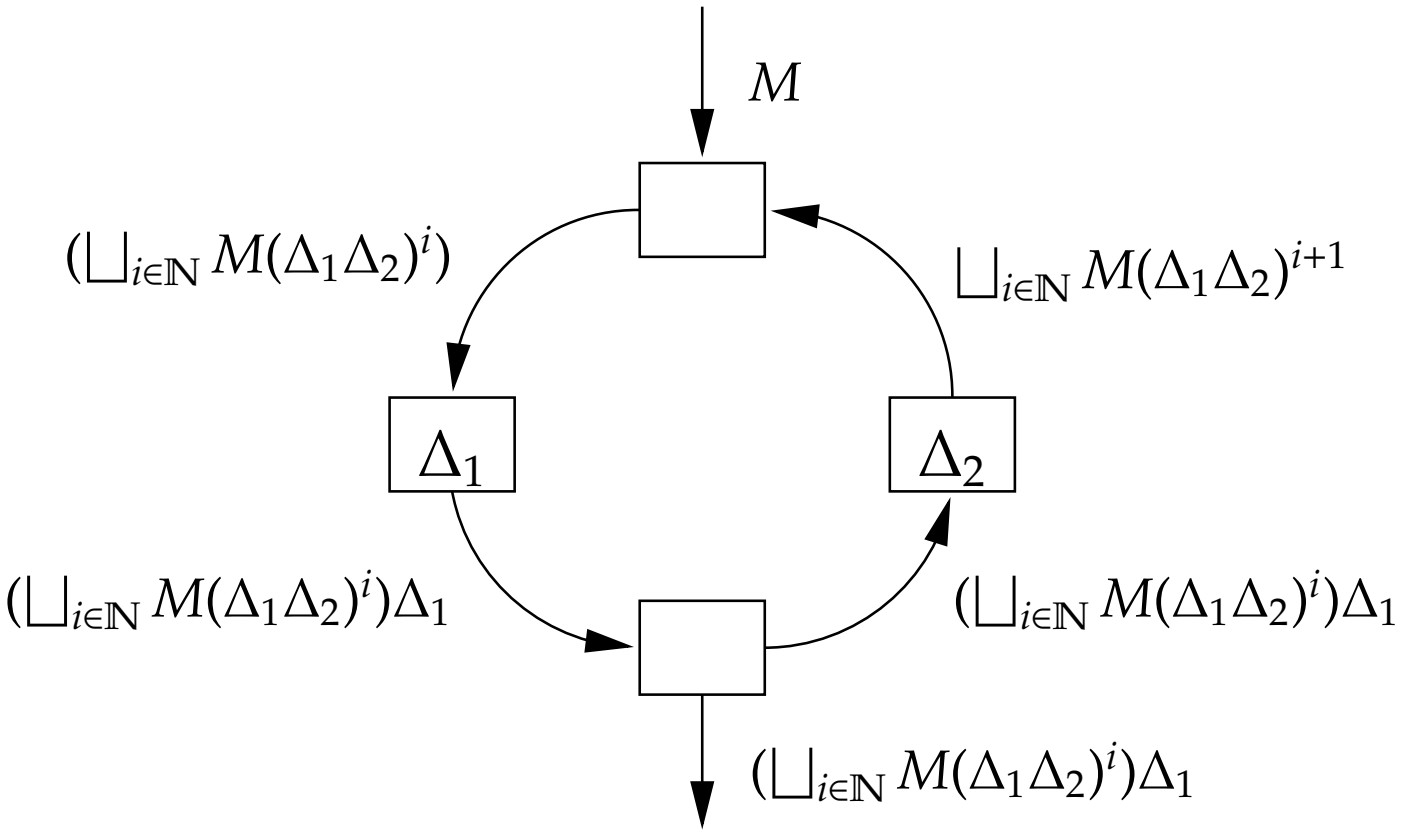
Example: a loop (without selective operators)



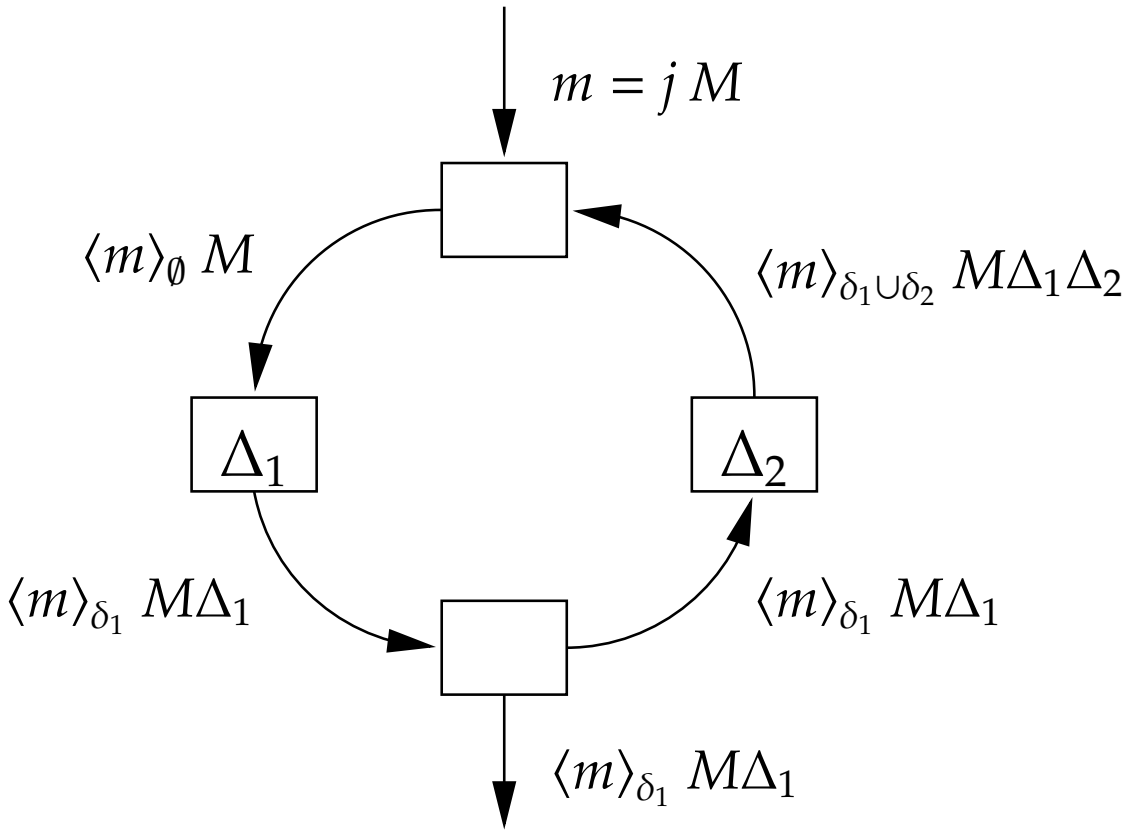
Example: a loop (without selective operators)



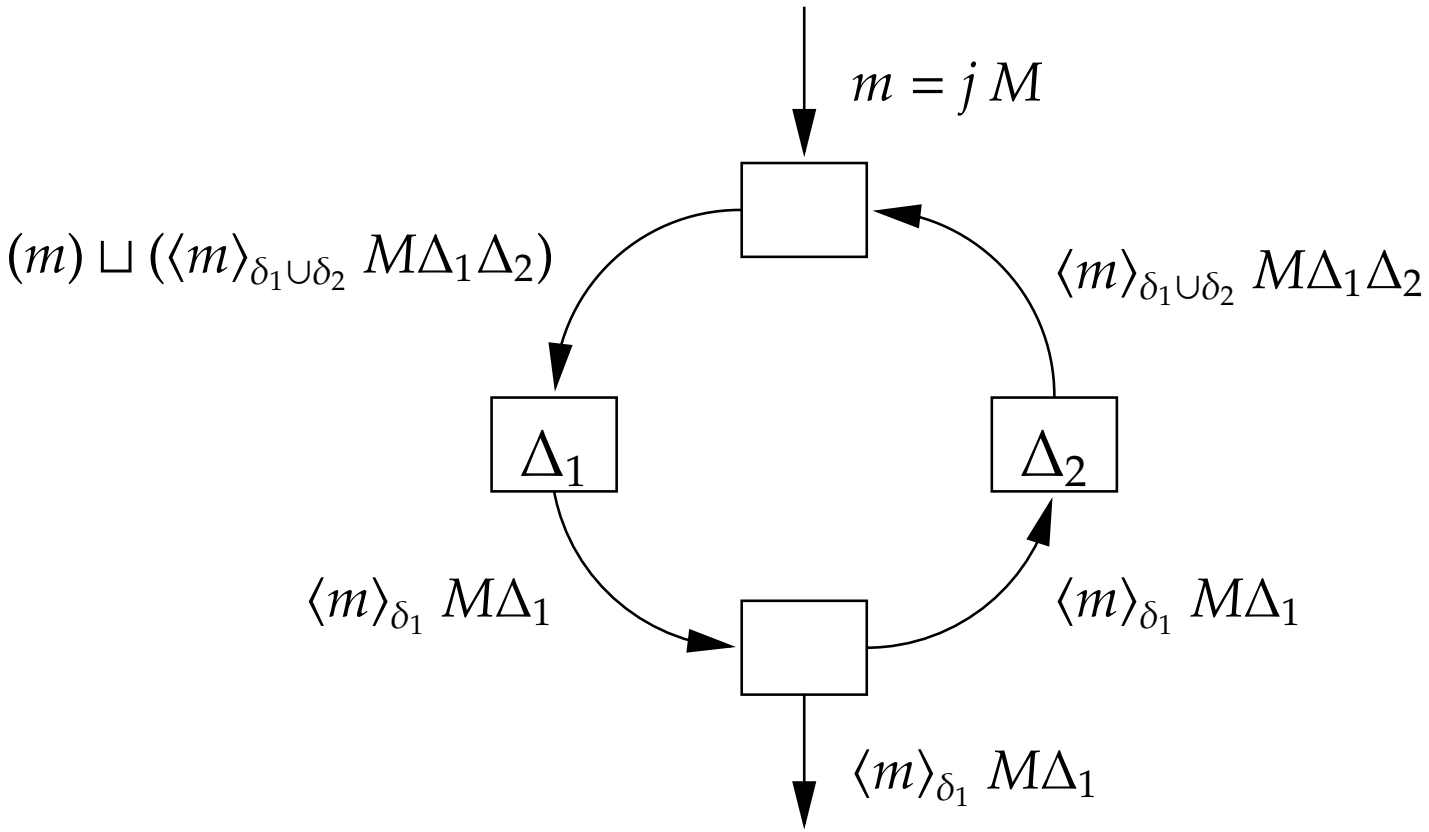
Example: a loop (without selective operators)



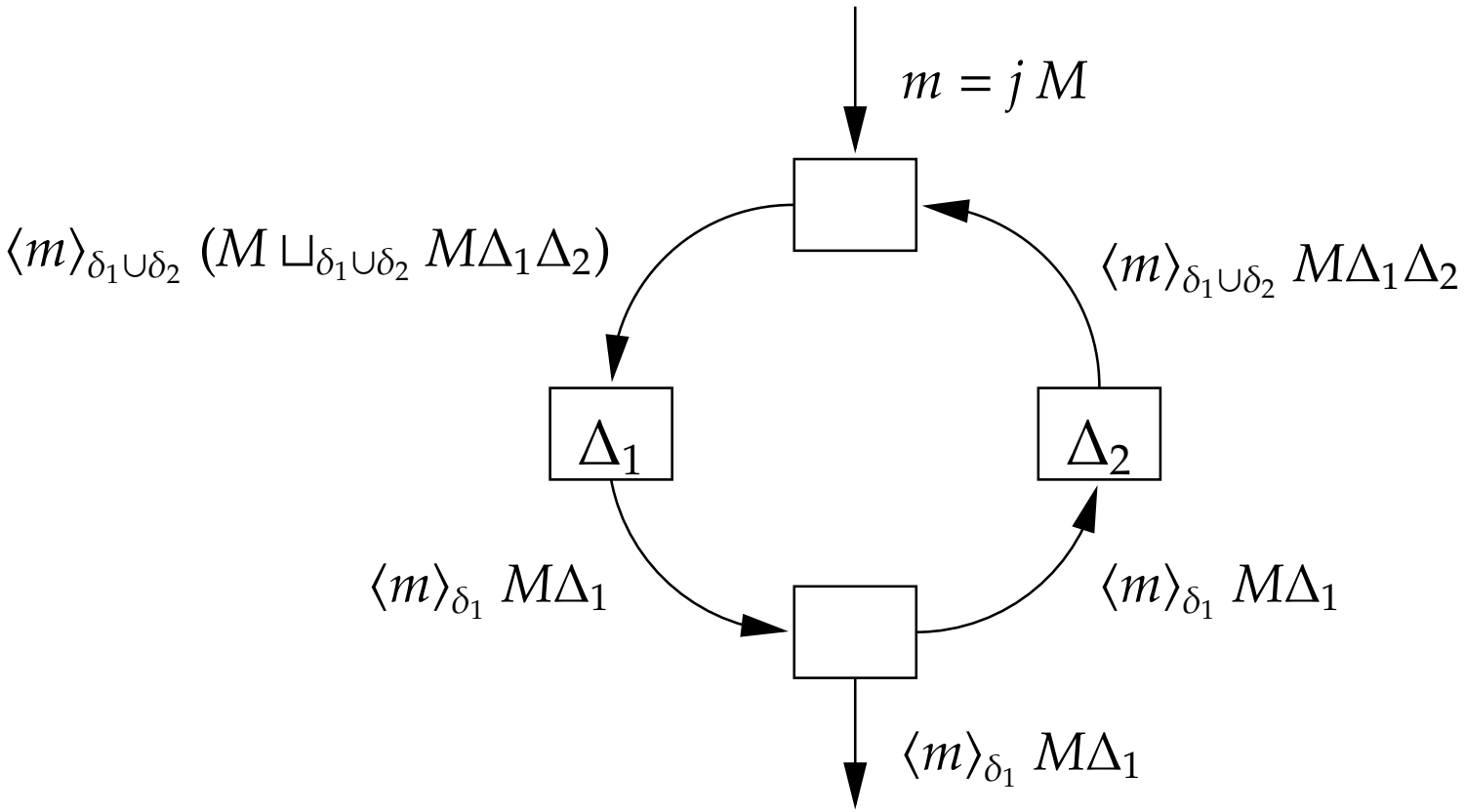
Example: a loop



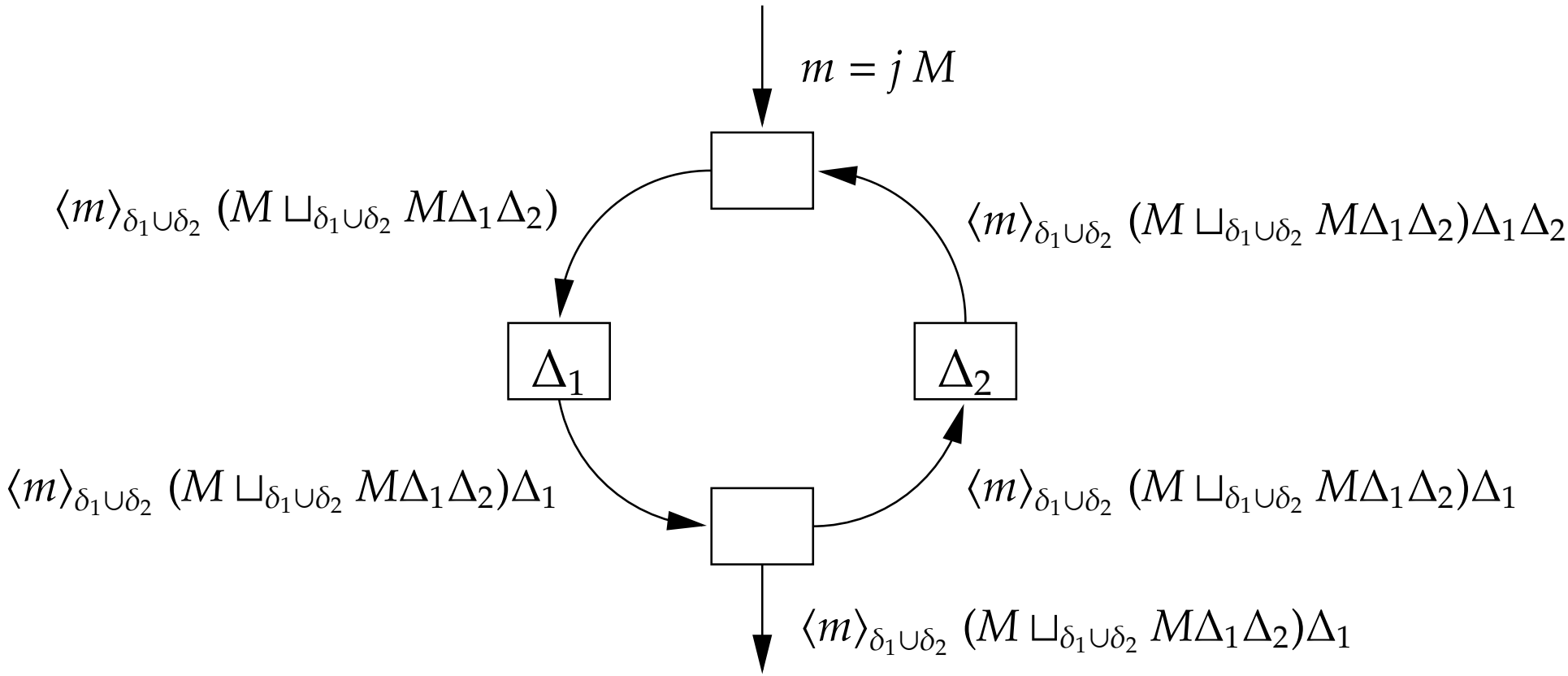
Example: a loop



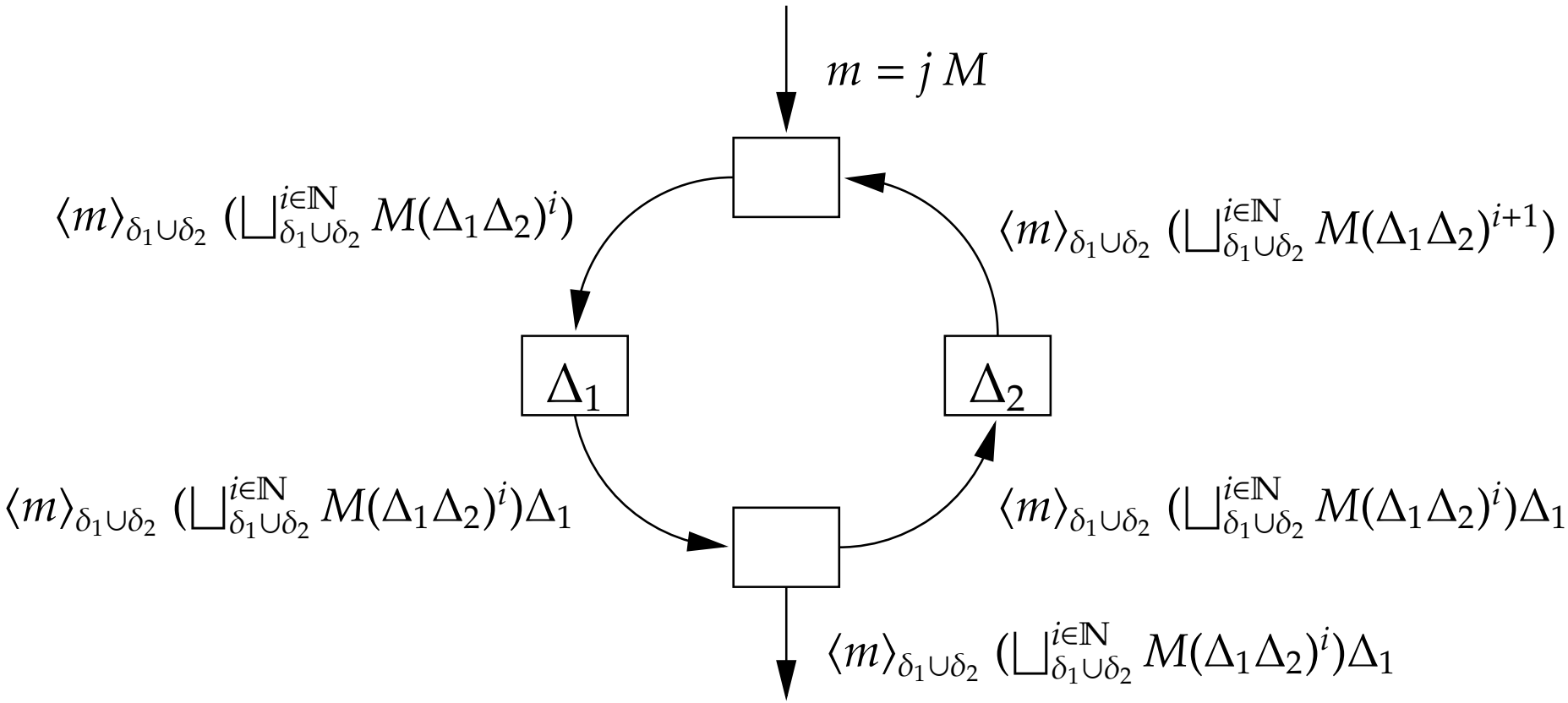
Example: a loop



Example: a loop



Example: a loop



Principle

For those that might join later with slight differences,
apply *wrap* to protect the untouched part before propagating.

Two *wrap*ing rules of ours:

1. *wrap* output of nodes that have more than two successors.
   (Since they might join later.)
2. *wrap* output of loopheads.
   (Since output will eventually join at loophead itself.)

- Localizing operators
  - Motivation
  - Journaling memory
  - Selective operators
  - Experiment results

Experiment[2]: r244

Name	Without (s)	With (s)	Improvement
atkbd.i	457.12	189.91	58.45%
eata_pio.i	975.49	690.47	29.21%
ip6_output.i	9147.53	6639.16	27.42%
xfrm_user.i	6618.20	4998.86	24.46%
usb-midi.i	3067.15	2380.09	22.40%
keyboard.i	981.56	785.77	19.94%
mptbase.i	13467.10	11290.20	16.16%
aty128fb.i	682.58	616.37	9.69%
af_inet.i	5189.83	5194.42	- 0.08%
cdc-acm.i	232.50	250.79	- 7.86%
		Average	19.98%
		Standard Deviation	18.09%

Table: Experiment results with r244

[2] All experiments were performed on a Pentium4 3.2GHz, 4GB main memory system.

Experiment: r332

Name	Without (s)	With (s)	Improvement
usb-serial.i	548.79	283.85	48.27%
usb-midi.i	3353.51	1841.59	45.08%
wavelan.i	10165.58	5741.34	43.52%
skge.i	49749.93	34551.73	30.54%
isdn_tty.i	19962.56	15711.49	21.29%
cdc-acm.i	172.66	144.22	16.47%
mptbase.i	17149.42	15013.55	12.45%
de4x5.i	14555.33	13301.8	8.61%
af_inet6.i	5336.33	5154.56	3.40%
i2o_proc.i	33556.94	35543.75	- 5.92%
		Average	15.62%
		Standard Deviation	16.01%

Table: Experiment results with r332 (includes local variable isolation)

Experiment: r332 (graph)

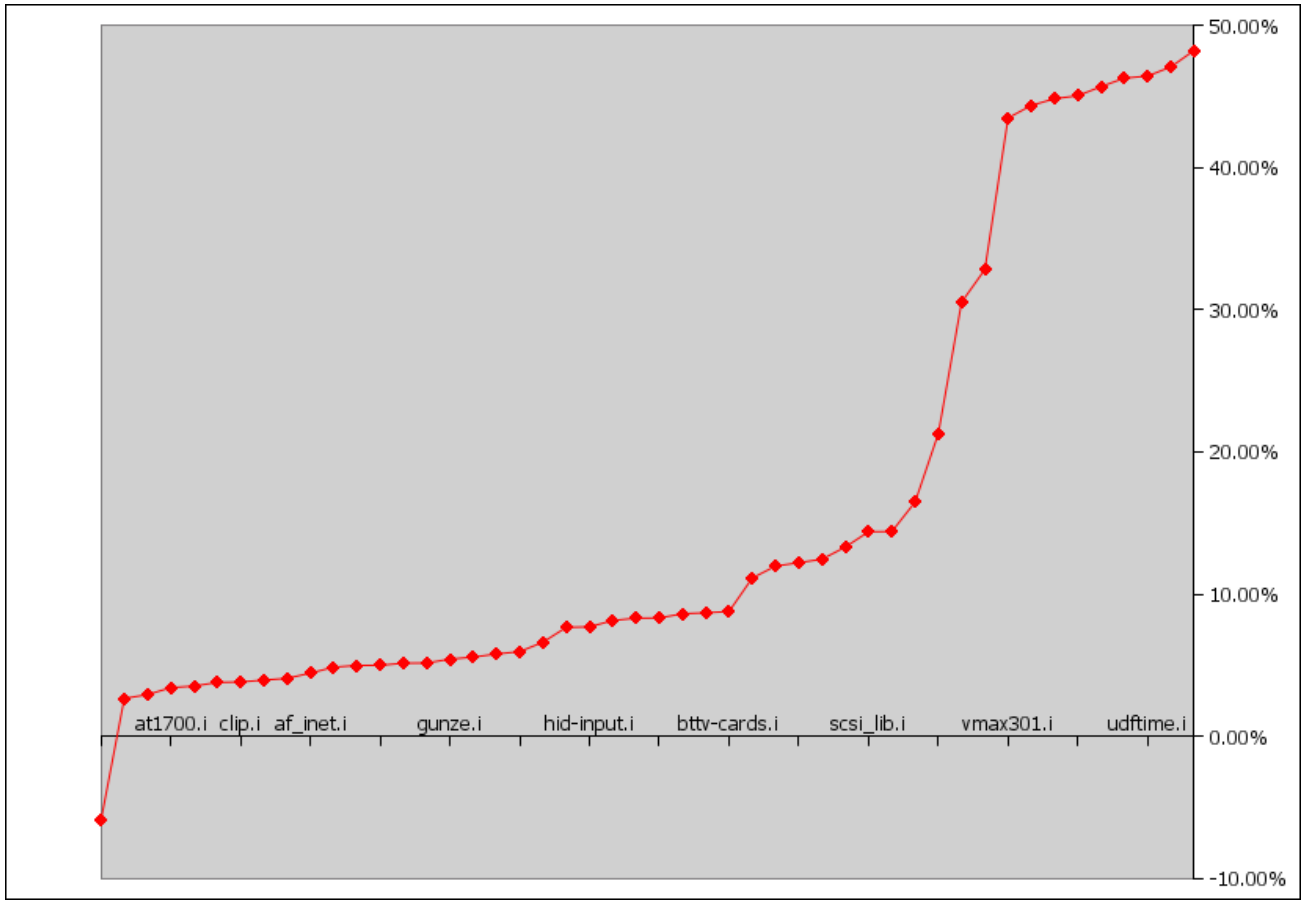


Figure: Experiment results with r332

Conclusion

- We could localize memory operators using
  - Idempotence of memory operators
  - Journal that keeps track of changes
- Time complexity reduced: $O(n \log n)$ to $O(k \log n)$
  where n: number of memory entries, k: number of updated ones
- Reduced 15% of the analysis time in average cases
- Almost twice as faster in many optimal cases

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

Local variable pollution

What's the approximated return value of `abs(n)`?

```
print(x) { ... }

abs(n) {
  if (n < 0) {
    n = -n;
    print("neg");
  } else {
    print("pos");
  }
  return n;
}
```

Problem

Join and widening are applied to all memory entries
even when they are not reachable from that point,
e.g. other procedure's local variables.

Solution

Make join and widening operators on memory
always operate on reachable entries only.

(But, computing reachability everywhere must be very expensive.)

Another solution

1. From every CALL, remove entries callee cannot reach.
2. From every EXIT, remove entries caller cannot reach.

(Less frequent, but still need to compute reachability.)

Our solution[3]

1. From every CALL, remove local variables whose address was never taken syntactically like `&x`.
2. From every EXIT, remove local variables. An exception in recursions: do not remove those holding return values.

(An effective way for removing entries definitely unreachable from any other procedure.)

[3] Joint work with Jaehwang

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

- Smart worklist algorithm
  Worklist algorithm
  Corrected mistakes

Worklist algorithm

$$W : Worklist \qquad n : Node \qquad old, new : Mem$$
$$G : Graph \qquad T : Table \qquad D : Dump$$
$$\mathcal{F}_n : Graph \times Mem \times Dump \rightarrow Graph \times Mem \times Dump$$

input$(W, (G, T, D))$
until $(isempty(W))$
  $n := choose(W);\ old := T(n)$
  $(G, new, D) := \mathcal{F}_n(G, \bigsqcup\{T(n') \mid n' \in pred(G, n)\}, D)$
  if $(isloophead(G, n))\ new := old \triangledown new$
  if $(new \not\sqsubseteq old)\ T(n) := new;\ add(W, succ(G, n))$
output(G, T, D)

Wait at join

- To minimize useless computations
- Two work lists: “active”, “waiting” (both LIFO)
- Choose nodes from “active” first if available.
- Choose from “waiting” otherwise.
- Insert join nodes[4] into “waiting”, others into “active”.

[4] whose number of predecessors is greater than 1

- Smart worklist algorithm
  Worklist algorithm
  Corrected mistakes

Mistake 1: stops after converged call

```
1  f(n) { ... }
2  main() {
3    int a[10];
4    f(1);
5    f(4);
6    f(2);
7    f(3); /* Problem: computation stops here */
8    a[1000] = 1;   /* no alarm */
9  }
```

Correction:
when adding CALL, also add its successive return points to the worklist.

Mistake 2: no return value from converged call

```
1  f(n) { ... }
2  main() {
3    int x, a[10];
4    f(1);
5    f(4);
6    x = f(2);
7    /* Problem: x is not defined here */
8    a[x] = 1;   /* alarms x is uninitialized */
9  }
```

Correction:
when adding ENTRY node of a procedure, also add predecessors of its EXIT to the worklist.

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

- Strong/weak updates
  Accuracy vs. soundness
  Another view of ”addresses”

Two updates

```
x = 1;
```

1. Strong (for accuracy)

$$M\{x \mapsto 1\}$$

2. Weak (for soundness)

$$M\{x \mapsto (M\ x) \sqcup 1\}$$

Question

When are we allowed to use “strong update”?

Example 1: variable

```
1  main() {
2    int a[10], x;
3
4    x = 100;
5    x = 1;
6    a[x] = 1;              /* value of x? */
7  }
```

Example 2: array element

```
1  main() {
2    int a[10], idx[2];
3
4    idx[0] = 100;
5    idx[1] = 1;
6    a[idx[1]] = 1;          /* value of idx[1]? */
7  }
```

Example 3: field of structure

```
1  main() {
2    int a[10]; struct { int l, r; } pair;
3
4    pair.l = 100;
5    pair.r = 1;
6    a[pair.r] = 1;          /* value of pair.r? */
7  }
```

Example 4: same field of structure

```
1  main() {
2    int a[10]; struct { int l, r; } pair;
3
4    pair.r = 100;
5    pair.r = 1;
6    a[pair.r] = 1;          /* value of pair.r? */
7  }
```

Example 5: same field of different structures

```
1  main() {
2    int a[10]; struct { int l, r; } pair[2];
3
4    pair[0].r = 100;
5    pair[1].r = 1;
6    a[pair[1].r] = 1;      /* value of pair[1].r? */
7  }
```

Example 6: dynamically allocated cell

```
1  main() {
2    int a[10], *ptr = malloc(sizeof(int));
3
4    *ptr = 100;
5    *ptr = 1;
6    a[*ptr] = 1;              /* value of *ptr? */
7  }
```

Example 7: dynamically allocated cells

```
1  main() {
2    int a[10], *ptr = malloc(sizeof(int) * 10);
3
4    *ptr++ = 100;
5    *ptr = 1;
6    a[*ptr] = 1;              /* value of *ptr? */
7  }
```

Example 8: pointing more than one

```
1  main() {
2    int a[10], *ptr, x = -1, y = 1;
3    if (...) ptr = &x; else ptr = &y;
4    *ptr = 100;
5    *ptr = 1;
6    a[*ptr] = 1;            /* value of *ptr? x? y?  */
7  }
```

Answer

We can use “strong update” only when
no other concrete execution can take place.

Our approach

Two types of address:

- "variable" for program variables, and
- "dynamic" for dynamically allocated addresses, array elements, and structure fields.

Use strong update only with singleton "variable" address sets.

- Strong/weak updates
  Accuracy vs. soundness
  Another view of ”addresses”

Set of addresses

"Abstract address" is a set of atomic abstract addresses.

Each atom means a set of concrete addresses,
and the meaning of an abstract address is the union of them.

Set as a map

- Set as a map

$$
\begin{array}{rcl}
(S) & : & S \to \{0 \sqsubset 1\} \\
(S)\, x & = & \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}
\end{array}
$$

- Union, intersection: pointwise join, meet

$$
\begin{array}{rcl}
(S \cup T) & = & (S) \sqcup (T) \\
(S \cap T) & = & (S) \sqcap (T)
\end{array}
$$

Three valued set

- Assign $\frac{1}{2}$ to an element if membership is unsure

$$(S) \quad : \quad S \rightarrow \{0 \sqsubset \tfrac{1}{2} \sqsupset 1\}$$

- "Three valued address set" will enable us to use strong updates for more cases, e.g. dynamic addresses.
- This might sound like "Three Valued Logic Analysis"

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999.

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

- Context sensitivity
  Why we need
  How we can

Procedure abstraction

- Airac5's analysis is context *insensitive*.
- That is, we consider a single approximation for body of a procedure by joining all memory states from its call sites.
- (I believe) It's an **over approximation** which not only decreases the accuracy but also increases cost.

Example: mymalloc

```
1  int *mymalloc(int n) {
2    int *p = malloc(n * sizeof(int));
3    return p;
4  }
5
6  f() {
7    int *a = mymalloc(10);
8    a[4] = 37;
9  }
10 g() {
11   int *b = mymalloc(20);
12   b[17] = 23;              /* false alarm */
13 }
```

Inlining?

- **No!**
  It's not a good solution if you're talking about inlining from the root of the call tree.
- Airac5 is able to inline top-down but it's practically useless due to extremely high cost.

- Let's see why top-down approach isn't appropriate.

Example: call graph of GNU grep 2.5.1

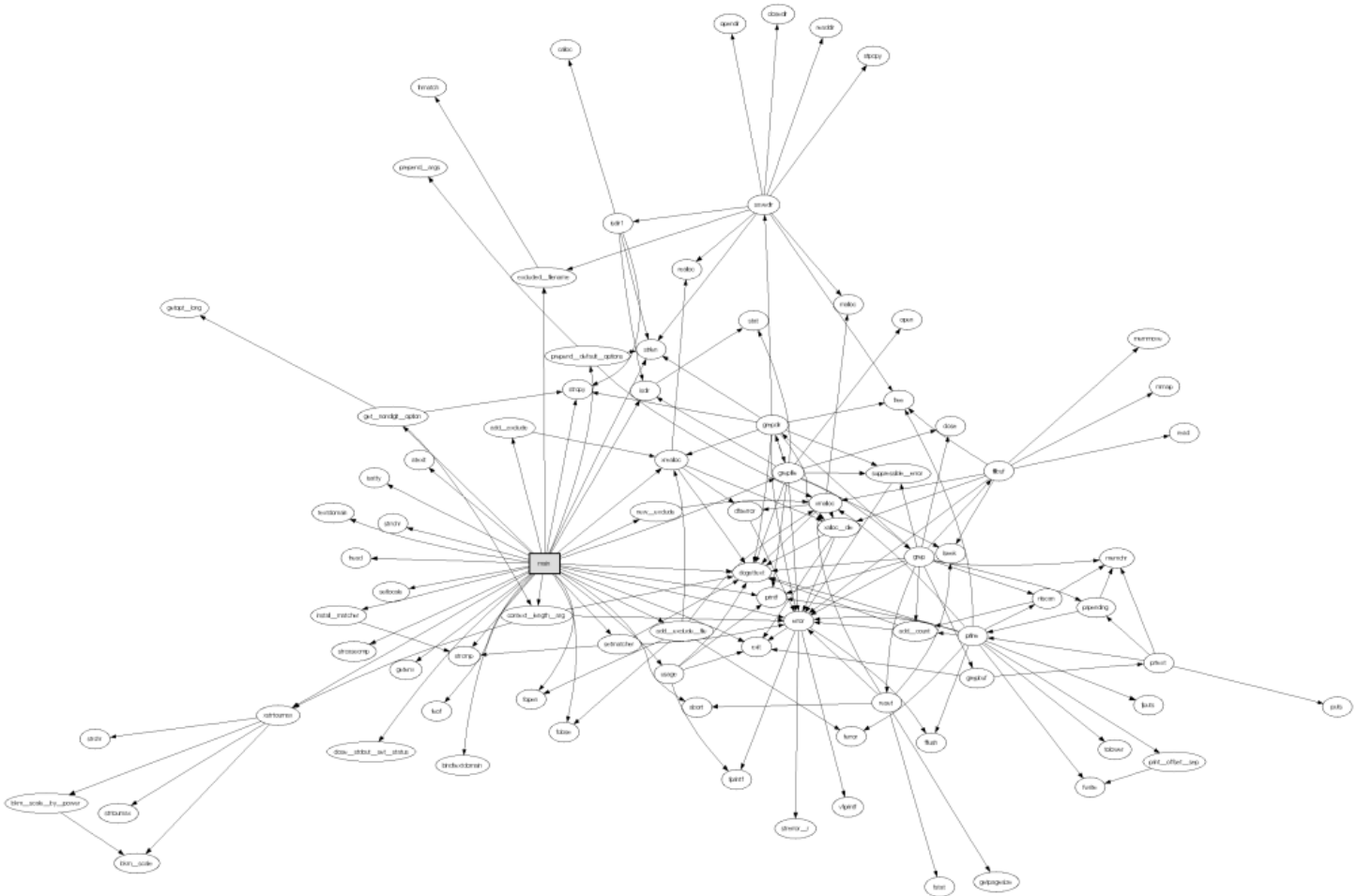


Figure: Procedures reachable from `main()`

Example: call graph of GNU grep 2.5.1 (magnified)

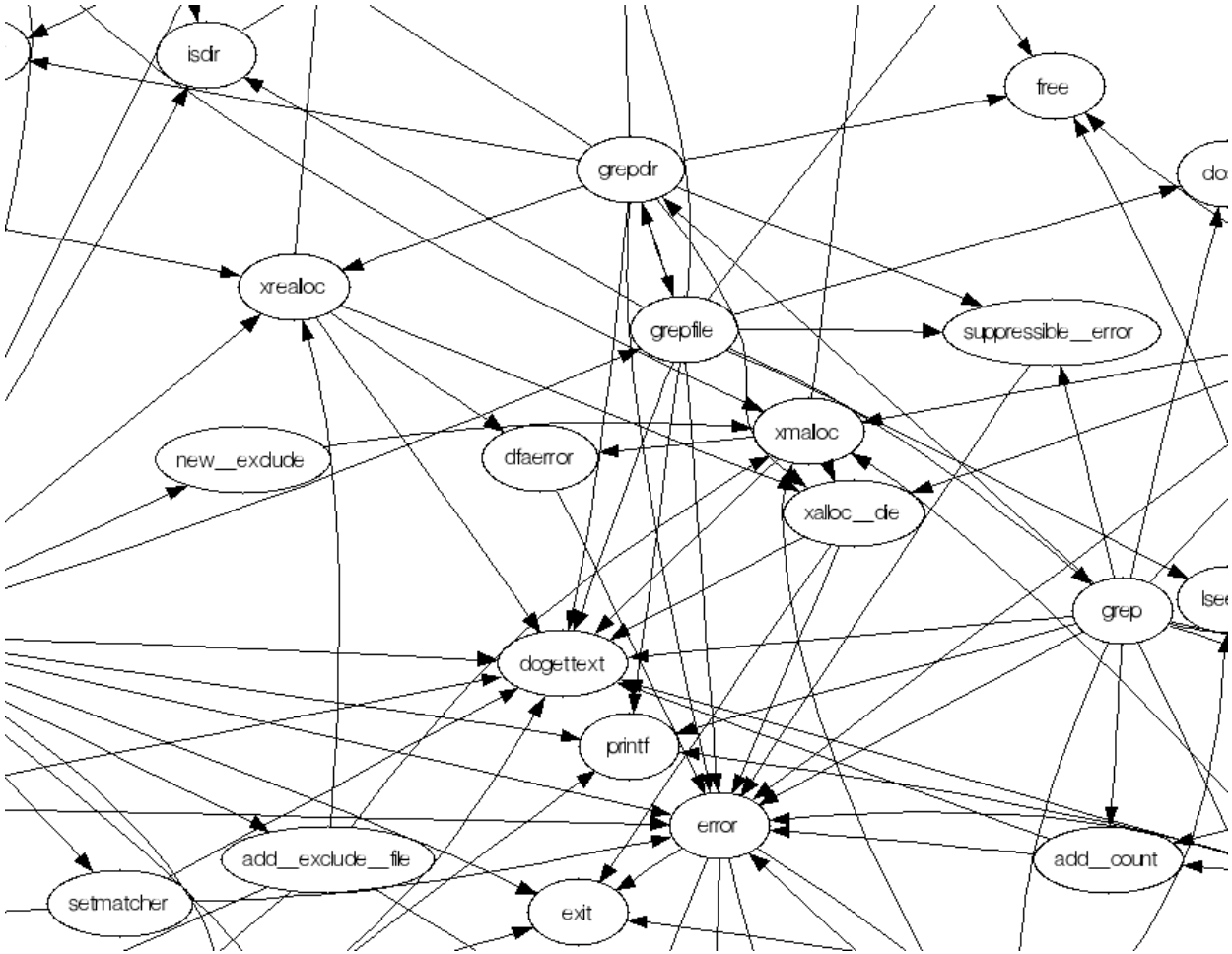


Figure: Important procedures called from many places

- Context sensitivity
  Why we need
  How we can

Inlining from the leaves

- Bottom-up approach
- Since we are mostly interested in procedures that call `malloc` or `realloc`, `free` rather than others, e.g. `xmalloc`, `xrealloc`.

Lightweight inlining

- No need to copy the graph (code).
- Just think them as another program point with different context.

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

Free variables

- “Real programs always have free variables”
- Lack of source code:
  proprietary library, development policy, etc.

- Airac5 assumes
  - undefined procedure makes no side effect
  - undefined variables can have any value,
    e.g. “extern” global variable, parameters of library function

Any nice way?

- Defining every possible free variables, i.e. closing the program prior to analysis is practically impossible.

- "Dynamic assumption"[5]:
  When we meet a free variable during the analysis,
  1. generate a label for it,
  2. guess its relation with others by looking at the code itself, i.e. by continuing the analysis as usual.

  Finally, show the generated assumptions(labels) along with the analysis result.

[5] Original idea developed in many discussions with Yungbum

- Localizing operators
- Isolating local variables
- Smart worklist algorithm
- Strong/weak updates
- Context sensitivity
- Handling free variables
- Other Thoughts

Examining the analysis results

After having some experiences of classifying alarms of Airac5 and Mairac, I thought:

- Too many alarms with insufficient information
- Time consuming, labor intensive
- Requires very little creativity or humanity
- Painful job for human beings
- Machines are better at it

and concluded:

We need "automatic refinement".

Refinement by gradually increasing sensitivities

After computing an approximation and gathering alarms as usual, repeat these steps until alarms don't change:

1. Increase sensitivities using information from the alarms:
   - Context sensitivity:
     distinguish all call sites to contexts of current alarms and
   - Path sensitivity:
     distinguish the paths between each alarm point and its nearest preceding join point.
2. Compute an approximation and gather alarms.

Analyzing very large programs

- Distributed fixpoint computing
  - more workers, earlier results
- External storage
  - To jump over the main memory limit
  - To handle the swapping problem ourselves
- Infinite refinement
  - Repeat automatic refinement until no alarm remains.
- Online user interface
  - Let user explore any part of the program with its approximation at anytime during the analysis process.

Thank you

Any questions, comments?
Any opinions, suggestions?

Let's talk about them.