

Incremental Flow Analysis through Computational Dependency Reification

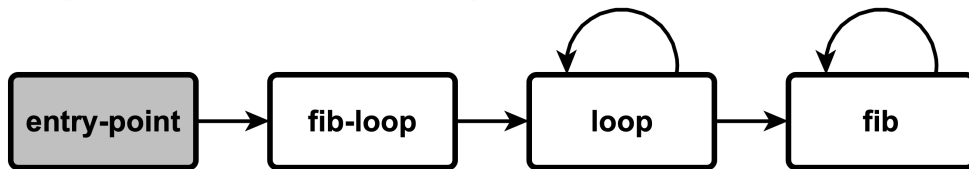
Jens Van der Plas, Quentin Stiévenart,
Noah Van Es, Coen De Roover

jens.van.der.plas@vub.be

SCAM 2020
September 27-September 28, 2020 - Adelaide, Australia

Static Analysis

E.g., control-flow graph



```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
    n
    (let ((fib-n-1 (fib (- n 1)))
           (fib-n-2 (fib (- n 2))))
      (+ fib-n-1 fib-n-2))))
```

```
(define (fib-loop n)
  (define (loop i)
    (if (< i n)
      (begin
        (display (fib i))
        (display " ")
        (loop (+ i 1))))
    (loop 0))
```

```
(fib-loop 10) ;; Entry point
```

Often, Static Analyses must be Fast

Better developer response [Harman et al., 2018]

Integration in IDE or CI-system

⇒ incremental analyses

Where Should Analysis Be Shown?

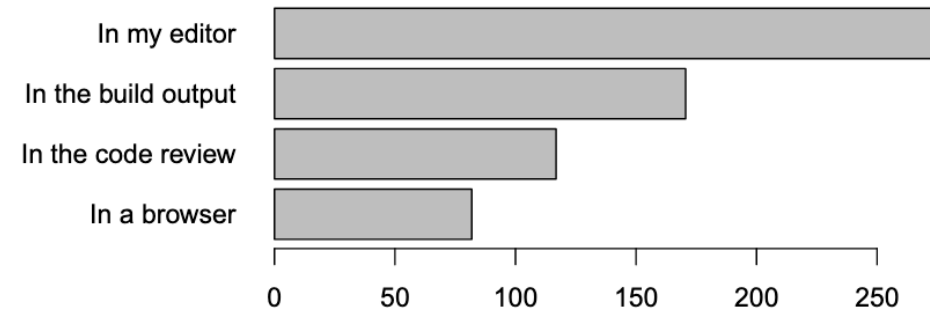


Figure 5: Where developers would like to have the output of program analyzers. [Christakis and Bird, 2018]

We introduce a novel, general incrementalisation approach for modular analyses.

Modular Analysis

[Cousot & Cousot, 2002]

Decomposes a program into *modules*

- functions, threads,...

Components: approximations of run-time equivalents

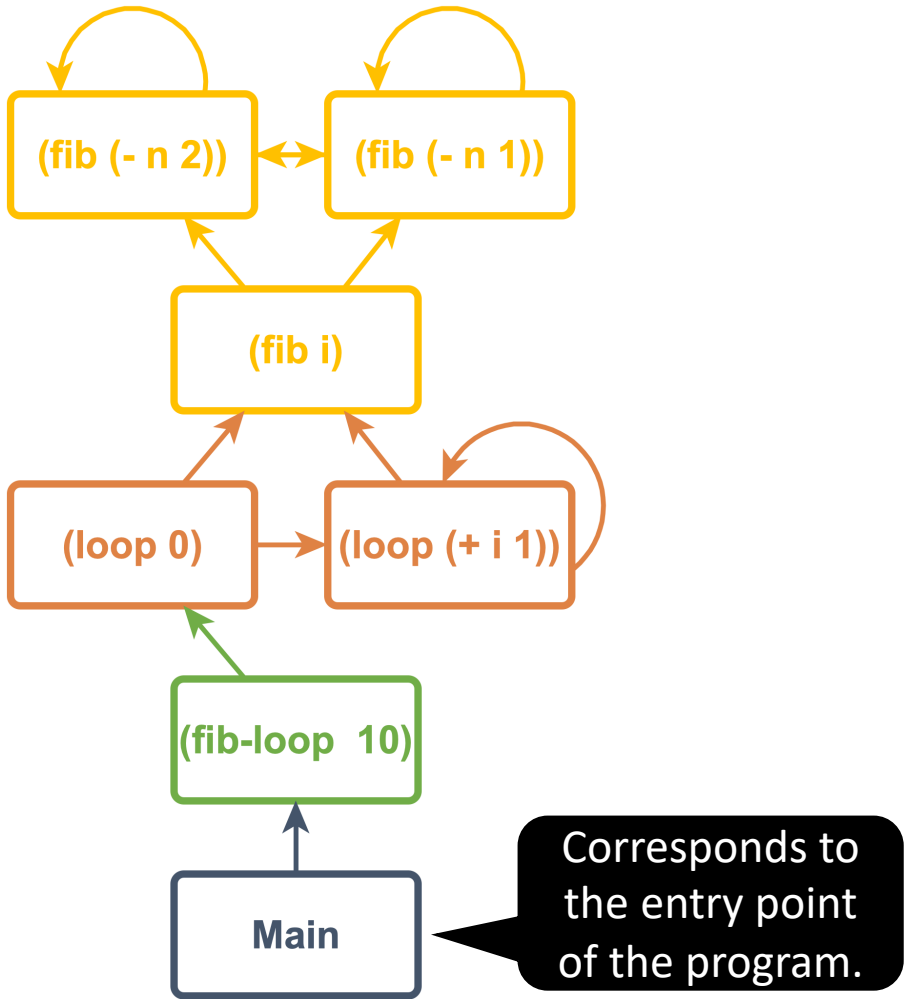
- Analysed in isolation
- Contain contextual information

```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))
```

```
(define (fib-loop n)
  (define (loop i)
    (if (< i n)
        (begin
          (display (fib i))
          (display " ")
          (loop (+ i 1)))))
  (loop 0))
```

```
(fib-loop 10) ;; Entry point
```

Modular Analysis



```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))
```

```
(define (fib-loop n)
  (define (loop i)
    (if (< i n)
        (begin
          (display (fib i))
          (display " ")
          (loop (+ i 1)))))
  (loop 0))
```

```
(fib-loop 10) ;; Entry point
```

Modular Analysis Algorithm

Contextual information: call expression

```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
    n
    (let ((fib-n-1 (fib (- n 1)))
           (fib-n-2 (fib (- n 2))))
      (+ fib-n-1 fib-n-2))))
```

```
| (fib 5) ;; Entry point |
```

Main

Modular Analysis Algorithm

Contextual information: call expression

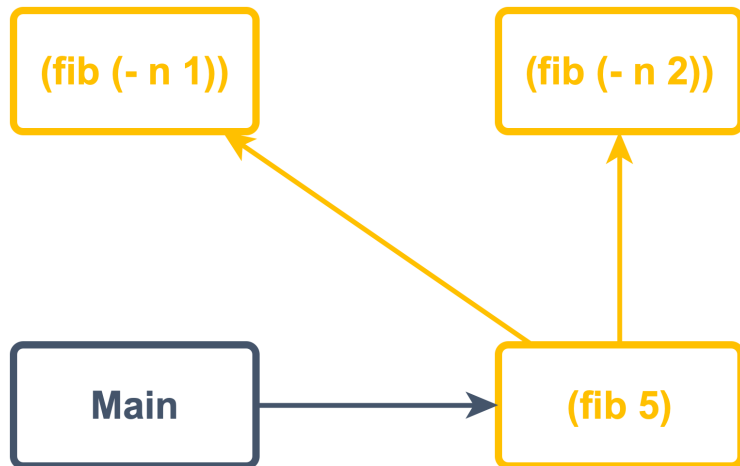
```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
    n
    (let ((fib-n-1 (fib (- n 1)))
           (fib-n-2 (fib (- n 2))))
      (+ fib-n-1 fib-n-2))))
```

```
(fib 5) ;; Entry point
```



Modular Analysis Algorithm

Contextual information: call expression

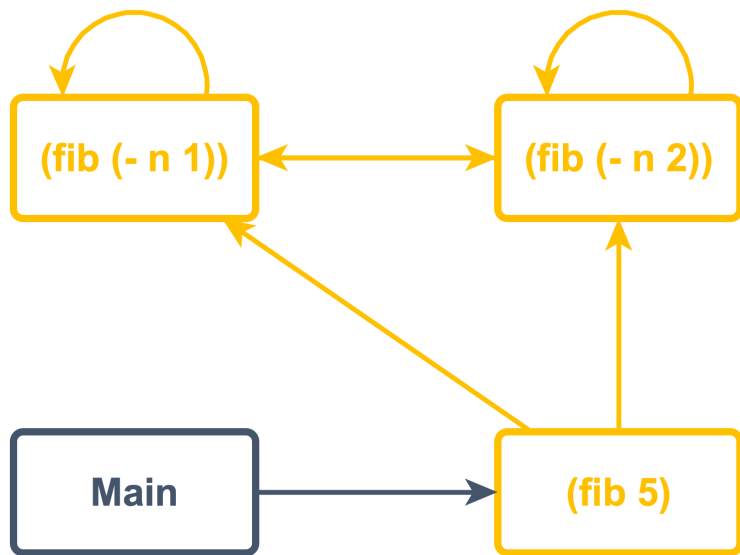


```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))
```

```
(fib 5) ;; Entry point
```

Modular Analysis Algorithm

Contextual information: call expression



```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))
```

```
(fib 5) ;; Entry point
```

Components may be reanalysed!

Dependency-driven

Computational Dependencies

Components are inter-dependent!

E.g., function-modular analysis:

- Argument values
- Return values
- Shared data (structures)

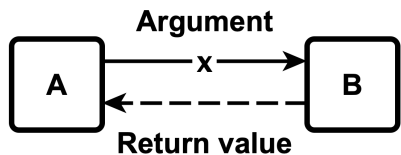
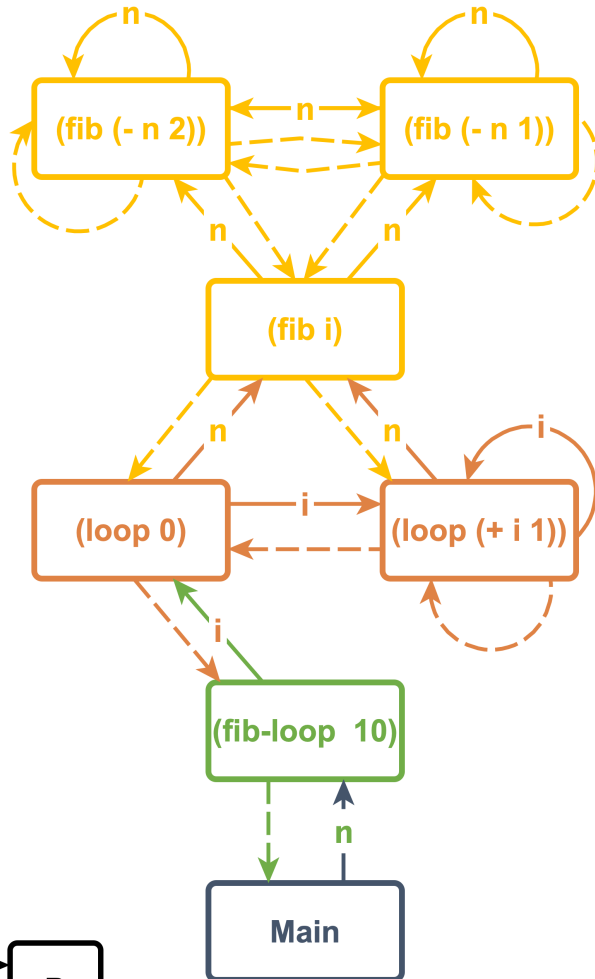
Reify dependencies between components

```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
    n
    (let ((fib-n-1 (fib (- n 1)))
           (fib-n-2 (fib (- n 2))))
      (+ fib-n-1 fib-n-2))))
```

```
(define (fib-loop n)
  (define (loop i)
    (if (< i n)
      (begin
        (display (fib i))
        (display " ")
        (loop (+ i 1))))
    (loop 0))
```

```
(fib-loop 10) ;; Entry point
```

Computational Dependencies



```
(define (fib n) ;; Simple Fibonacci
  (if (= n 0)
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))
```

```
(define (fib-loop n)
  (define (loop i)
    (if (< i n)
        (begin
          (display (fib i))
          (display " ")
          (loop (+ i 1)))))
  (loop 0))
```

```
(fib-loop 10) ;; Entry point
```

Change representation

Use of *change expressions*

- Computed through change analysis
- Manually specified

E.g. a bugfix

```
(define (fib n) ;; Simple Fibonacci
  (if (<change> (= n 0) (< n 2))
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))

(fib 5) ;; Entry point
```

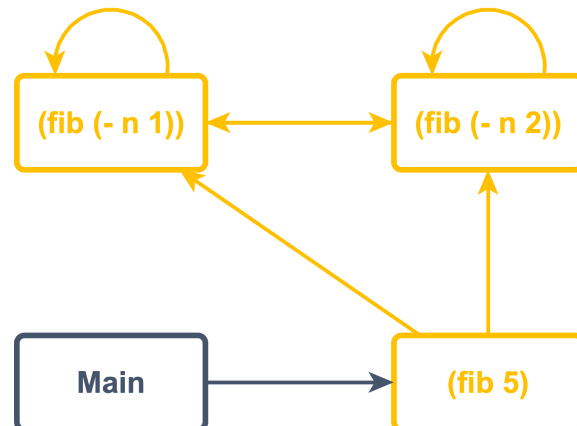
Incremental Update

Analysis state = value store σ :

```
{  
  var<fib>           → {#closure}  
  var<n>             → {Int}  
  var<fib-n-1>      → {Int}  
  var<fib-n-2>      → {Int}  
  ret<(fib 5)>       → {Int}  
  ret<(fib (- n 1))> → {Int}  
  ret<(fib (- n 2))> → {Int}  
  ret<Main>         → {Int}  
}
```

```
(define (fib n) ;; Simple Fibonacci  
  (if (< n 2)  
      n  
      (let ((fib-n-1 (fib (- n 1)))  
            (fib-n-2 (fib (- n 2))))  
        (+ fib-n-1 fib-n-2))))  
  
(fib 5) ;; Entry point
```

Components:
(dependencies omitted)



Incremental Update

Analysis state = value store σ :

```
{  
  var<fib>           → {#closure}  
  var<n>             → {Int}  
  var<fib-n-1>      → {Int, Real}  
  var<fib-n-2>      → {Int, Real}  
  ret<(fib 5)>       → {Int, Real}  
  ret<(fib (- n 1))> → {Int, Real}  
  ret<(fib (- n 2))> → {Int, Real}  
  ret<Main>         → {Int}  
}
```

```
(define (fib n) ;; Simple Fibonacci  
  (if (< n 2)  
      (<change> n (* n 1.0))  
      (let ((fib-n-1 (fib (- n 1)))  
            (fib-n-2 (fib (- n 2))))  
          (+ fib-n-1 fib-n-2))))
```

```
(fib 5) ;; Entry point
```

Monotonic
update: precision
is lost

Immediately affected components are reanalysed

Incremental Update

Analysis state = value store σ :

```
{
  var<fib>           → {#closure}
  var<n>             → {Int}
  var<fib-n-1>      → {Int, Real}
  var<fib-n-2>      → {Int, Real}
  ret<(fib 5)>       → {Int, Real}
  ret<(fib (- n 1))> → {Int, Real}
  ret<(fib (- n 2))> → {Int, Real}
  ret<Main>         → {Int, Real}
}
```

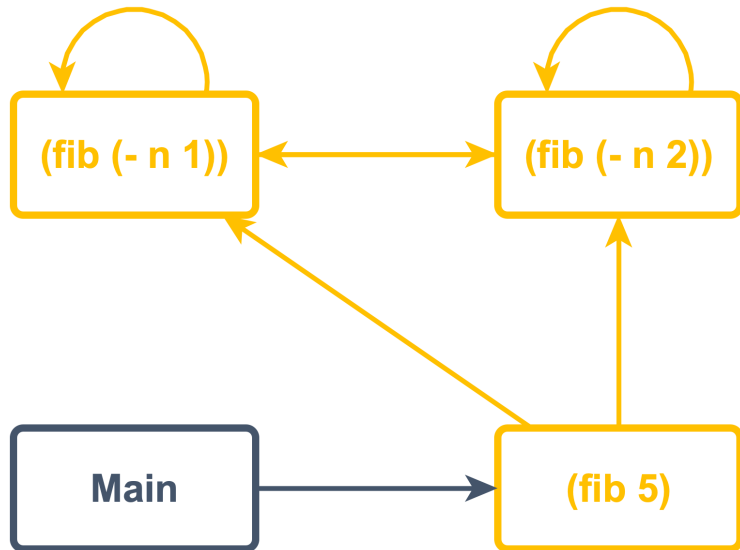
```
(define (fib n) ;; Simple Fibonacci
  (if (< n 2)
      (<change> n (* n 1.0))
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))
```

```
(fib 5) ;; Entry point
```

Dependencies cause indirectly affected components to be updated as well

Step 1: Change Impact Calculation

Find *directly affected* components



```
(define (fib n) ;; Simple Fibonacci
  (if (<change> (= n 0) (< n 2))
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
          (+ fib-n-1 fib-n-2))))

(fib 5) ;; Entry point
```

1. For every expression encountered, register the “current component” to the expression
`trackMap :: Map[Expr → Set[Comp]]`
2. Look up **updated expressions** in `trackMap` and collect components

Step 2: Update Analysis Results

1. Add the directly impacted components to the work list
2. Restart the inter-component analysis

Impact of changes is *bounded to impacted components*

Two Instantiations

1. Function-modular analysis for Scheme (as in example before)
2. Thread-modular analysis for concurrent Scheme

Both context-insensitive

- Other sensitivities are supported

Evaluation

Six metrics

- Analysis time
- Precision of analysis store
- Size of analysis store
- Number of components created
- Number of dependencies inferred
- Number of intra-component analyses performed

16 benchmark programs

- 6 for ModF
- 9 for ModConc

Benchmark	LOC	Description of the Sequential Program	Changes	#Changes
mceval-dynamic	246	Meta-circular evaluator for Scheme, executing a small Scheme program.	Changed the evaluator so procedures become dynamically scoped.	4
leval	378	Lazy Scheme evaluator, used to perform some list computations.	Changes to the evaluator so that only specific arguments are evaluated lazily.	11
multiple-dwelling (fine)	404	Evaluator for a non-deterministic Scheme, used to solve an allocation problem.	Fine-grained changes to the input for the evaluator.	3

```
(define (bound-expr var frame)
  (cond ((or (<change> (eq? (car frame) 'let) (tagged-list? frame 'let))
            (<change> (eq? (car frame) 'letrec) (tagged-list? frame 'letrec)))
        (cadr (assq var (cadr frame))))
        ((<change> (eq? (car frame) 'lambda) (tagged-list? frame 'lambda))
         not-constant)
        (else (error "ill-formed frame"))))
```

Evaluation

Function-modular

Thread-modular

ModF

ModConc

Benchmark	Initial Analysis [ms]	Full Reanalysis [ms]	Incremental Update [ms]	Δ	Benchmark	Initial Analysis [ms]	Full Reanalysis [ms]	Incremental Update [ms]	Δ
mceval-dynamic	226	124	72	-41.94%	mcarlo2	9	29	27	-6.90%
leval	1407	1971	489	-75.19%	pc	21	16	11	-31.25%
multiple-dwelling (fine)	8466	8822	2126	-75.90%	msort	117	151	194	+28.48%
multiple-dwelling (coarse)	3527	3533	15694	+344.21%	pps	421	423	1	-99.76%
peval	19753	17644	103	-99.42%	sudoku	86	90	62	-31.11%
nboyer	1397	1271	98	-92.29%	actors	1601	1595	354	-77.81%
machine-simulator	54124	57043	24093	-57.76%	stm	5384	5597	745	-86.69%
					crypt	7568	7351	2812	-61.75%
					crypt2	9315	10277	8340	-18.85%

✓ Faster on almost all benchmark programs

Evaluation

Function-modular

Thread-modular

ModF					ModConc				
Benchmark	Equally Precise	Less Precise	Less Precise [%]	Address Count (Δ)	Benchmark	Equally Precise	Less Precise	Less Precise [%]	Address Count (Δ)
mceval-dynamic	158	220	58.20%	10	mcarlo2	28	2	6.67%	1
level	187	389	67.53%	10	pc	35	4	10.26%	1
multiple-dwelling (fine)	851	0	0.00%	0	msort	27	9	25.00%	1
multiple-dwelling (coarse)	231	817	77.96%	198	pps	99	0	0.00%	0
peval	919	2	0.22%	0	sudoku	101	0	0.00%	0
nboyer	2115	17	0.80%	1	actors	136	0	0.00%	0
machine-simulator	1676	14	0.83%	7	stm	156	0	0.00%	0
					crypt	141	3	2.08%	3
					crypt2	140	6	4.11%	6

! Not always as precise, but can be remedied

Overview

- ✓ General incrementalisation approach for modular analyses
- ✓ Does not require fixed call-graph known upfront
- ✓ Not tailored to specific analysis
- ✓ Faster on almost all benchmark programs: lightweight & bounds change impact
- ! Not always as precise, but can be remedied

Questions

1. How much variance is acceptable in the reduction of the analysis time? (E.g., we obtained values in [-99.76%, +344.21%.])
2. What is the best way to obtain program changes for the evaluation of the incremental analysis? E.g., would the audience be willing to accept a change logger in their IDE?