

Hancock: A Language for Extracting Signatures from Data Streams

Corinna Cortes
Kathleen Fisher
Daryl Pregibon
Anne Rogers
Fred Smith (Cornell)

AT&T Labs–Research



AT&T Infolab

Networks:

- Long Distance
- Wireless
- Data: Frame Relay, ATM, IP
- Cable

Applications:

- Manage Network
- Prevent/Detect Fraud
- Understand Customers

Challenge:

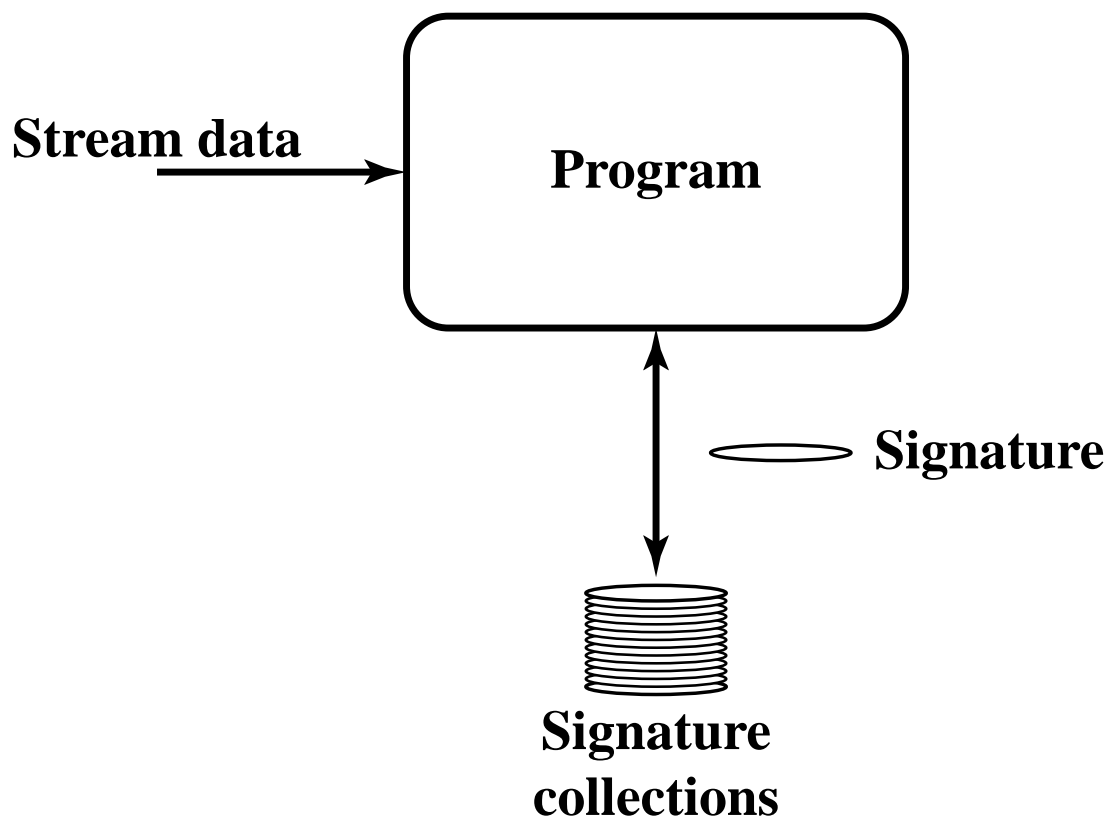
To convert all this data into useful information.



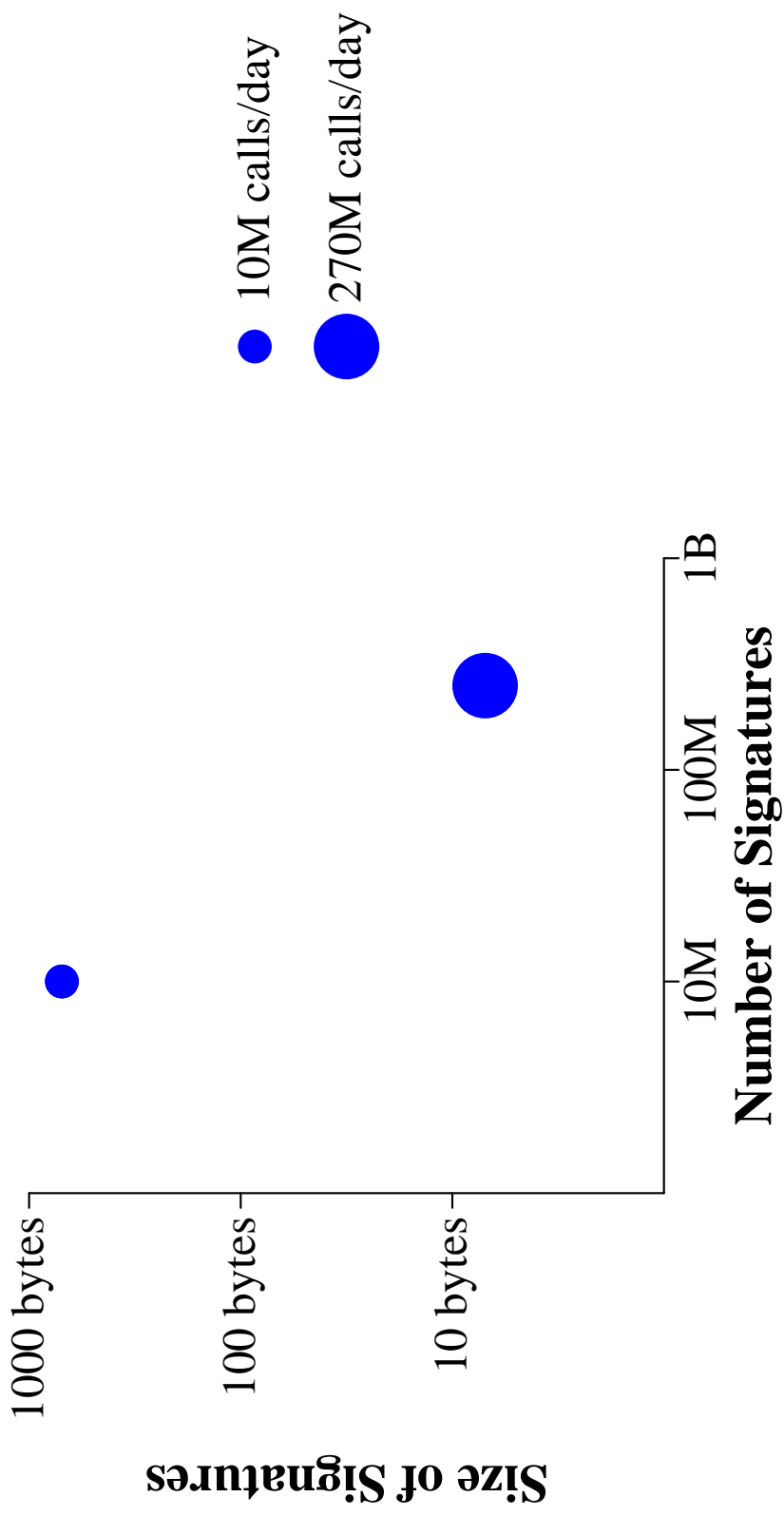
Problem statement

Efficiently managing communications-scale data requires substantial programming expertise.

Example: Extracting **signatures** (evolving profiles of customer calling behavior) from long distance data.



Scale!



Our approach

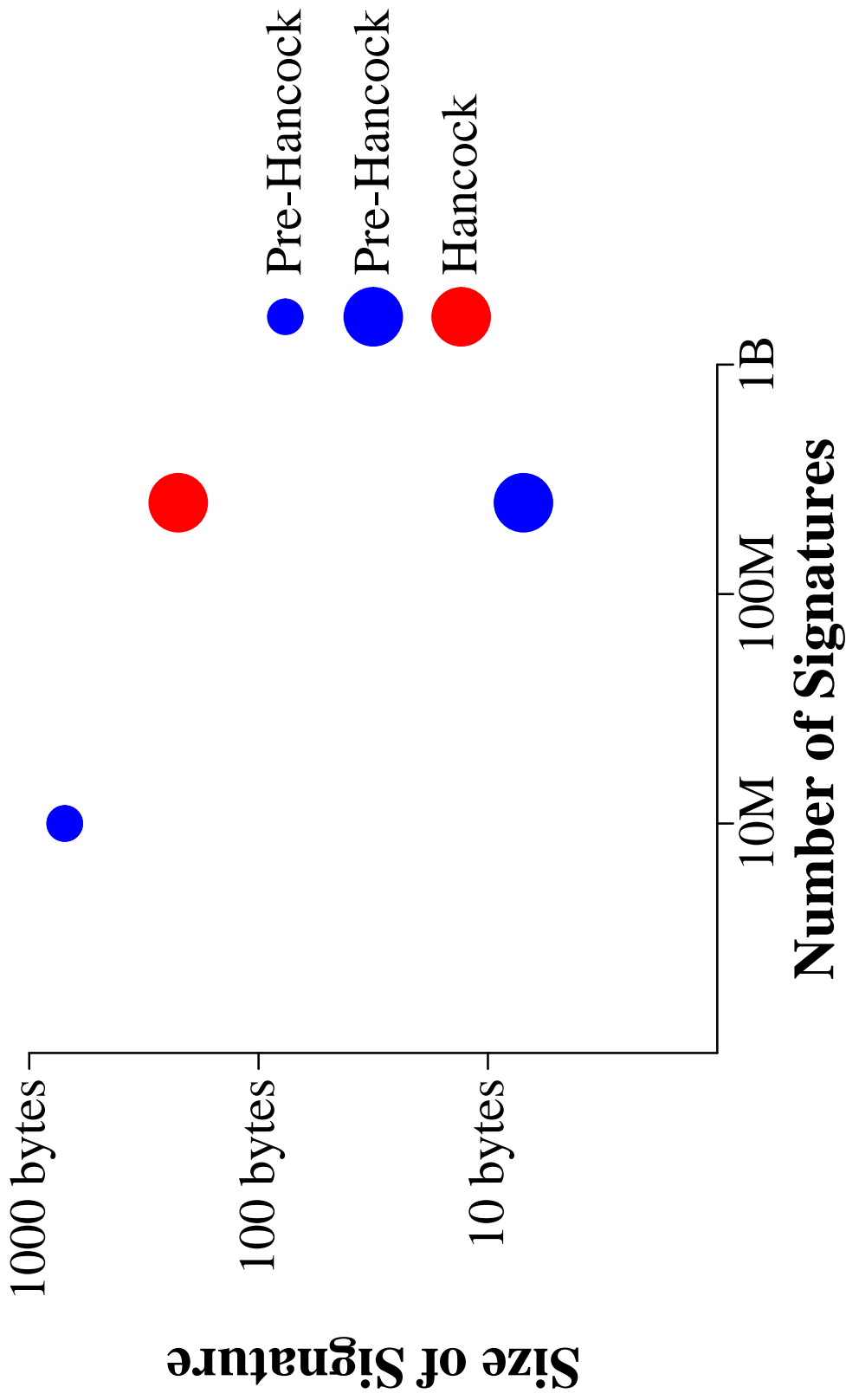
Hypothesis: A language could reduce the programming expertise required to compute with large data streams.

Experiment: Signature applications.

Results:

- Identified key abstractions for computing signatures.
- Generalized them to **any** large, homogeneous data source.
- Embedded them in **Hancock**, a C-based language.

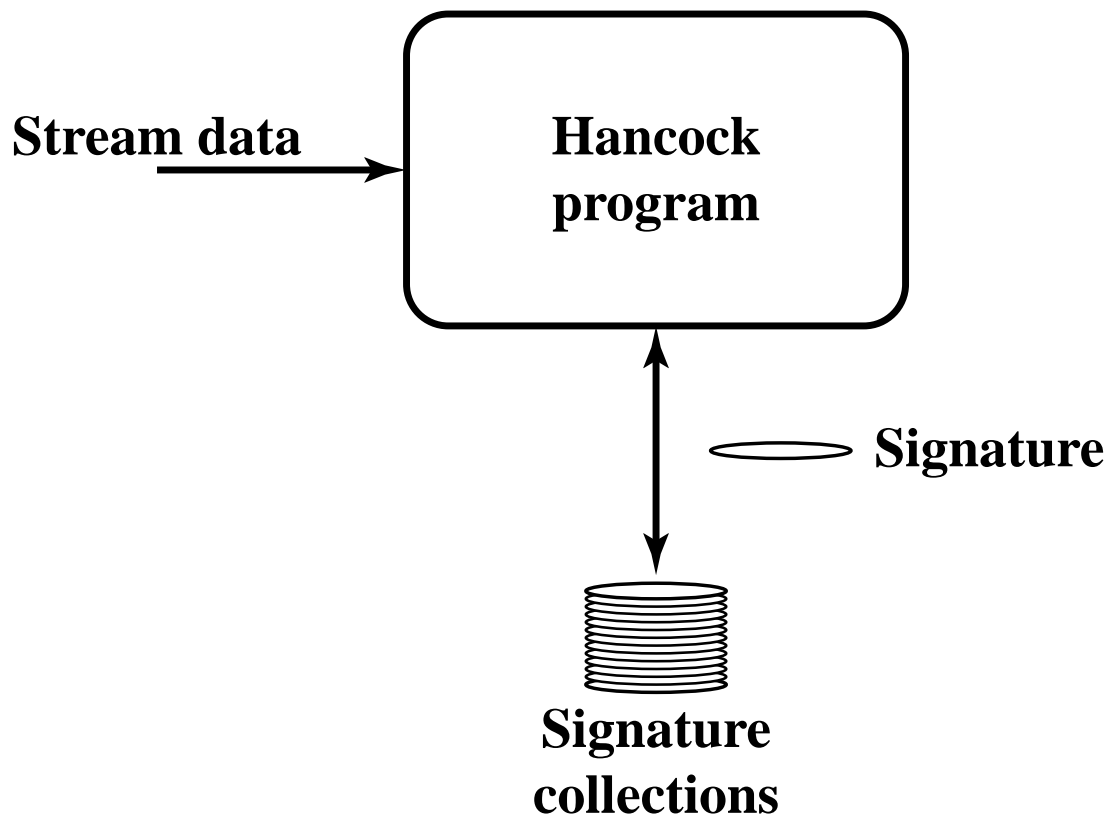
Concrete results



Hancock overview

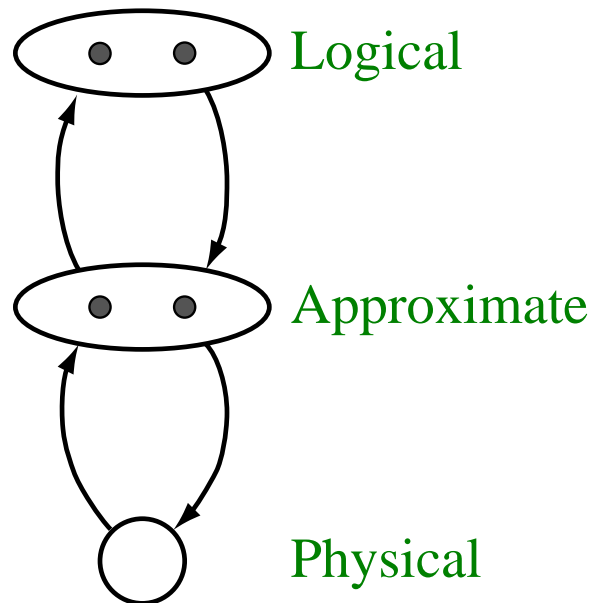
Wireless Messages:

Approximate number of calls to/from each mobile phone.



Logical, approximate, physical signatures

Issue: Using multiple views to reduce I/O:



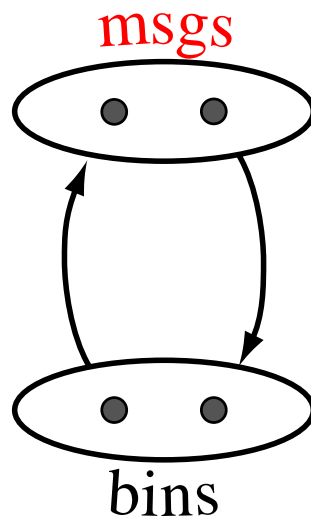
Goals:

- Group **logical** and **approx.** views.
- Clarify usage.

Hancock Views:

- Specify two views of data.
- Switch between them easily.

Wireless message profile



```
view (bins, msgs) {  
    char <=> int in;  
    char <=> int out;  
    bins(m) { return msgs_to_bins(m); }  
    msgs(b) { return bins_to_msgs(b); }  
}
```

Sample use: $m = b\$msgs;$

Signature collections

Issue: Need to associate signatures with keys efficiently.

Goal: Separate use from implementation.

Hancock Maps:

- Direct addressing.
- Customized (compressed) **physical** format.
- Programmable defaults.

Wireless messages map

```
map cellMsgs_m {  
  key pn_t;  
  value bins;  
  default {0, 0};  
  compress binsSqueeze;  
  decompress binsUnsqueeze;  
} cm;
```

Map indexing:

```
b = cm<:mpn:>;  
  ...  
cm<:mpn:> = b;  
  ...  
m = cm<:mpn:>$msgs;
```

Logical and physical streams

Issue: Physical representation is (often) highly encoded.

Goals:

- Isolate the physical representation.
- Focus on the logical representation.

Hancock streams: Specify

- Physical representation.
- Logical representation.
- Translation between them.

Wireless data source

```
typedef struct {  
    pn_t origin;  
    pn_t dialed;  
    ...  
    char isOrigCell;  
    char isDialCell;  
} awsLog_t;
```

```
char getvalidAWS(awsPhy_t *pc, awsLog_t *c);
```

```
stream AWS_s {  
    getvalidAWS : awsPhy_t => awsLog_t;  
};
```

Consuming streams

Issue: Combining complex event detection and response code obscured meaning.

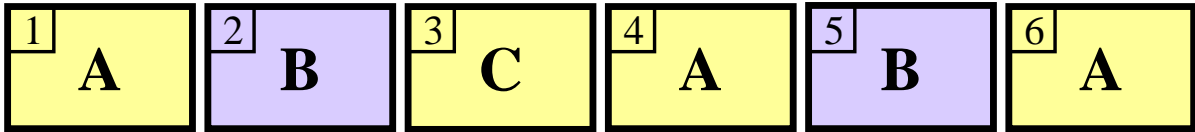
Goal: Reveal essential computation.

Hancock's Iterate Statement:

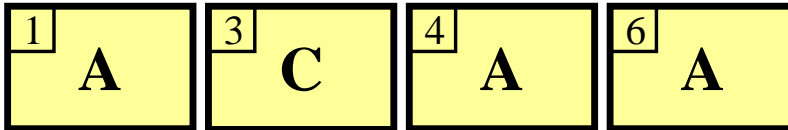
- Separates event detection from event response.
- Generates the scaffolding.

Iterate

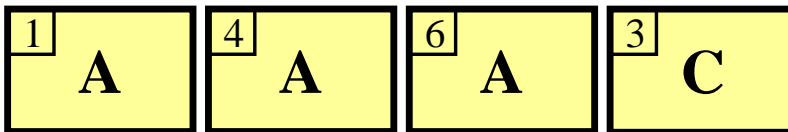
Initial stream of records:



Filter to remove unneeded records:



Sort to ensure access locality:



Structure computation using **events**.

Wireless messages outgoing phase

```
int isOriginCellCall(awsLog_t *lc);
```

```
iterate
```

```
  ( over calls  
    filteredby isOriginCellCall  
    sortedby origin  
    withevents originDetect ) {
```

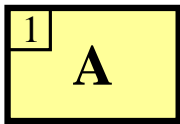
```
    event response code
```

```
};
```

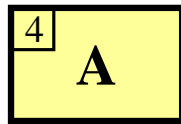
Representing events

Hancock's multi-union (**munion**): a set of labels and associated signatures.

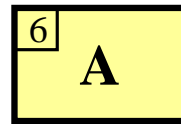
```
munion line_e {:  
  pn_t      begin_line,  
  awsLog_t  call,  
  pn_t      end_line  
:};
```



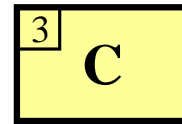
begin_line(A)
call



call



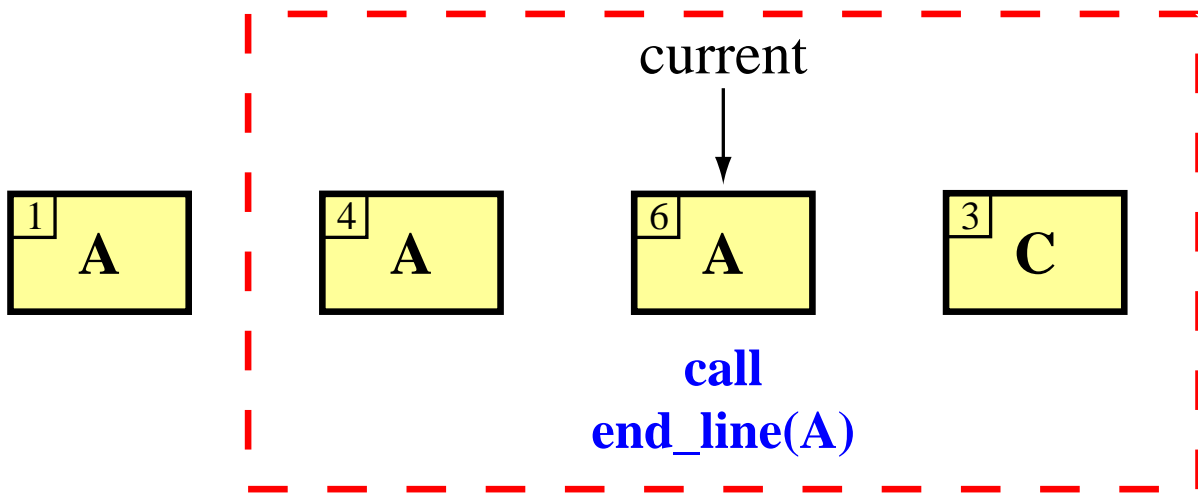
call
end_line(A)



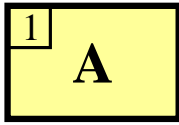
begin_line(C)
call
end_line(C)

Event detection function

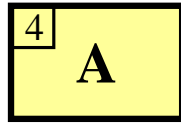
Takes a **window** onto a stream and returns a **munion**.



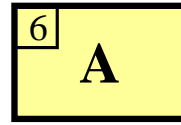
Event response code



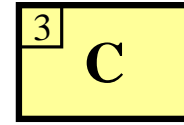
begin_line(A)
call



call



call
end_line(A)



begin_line(C)
call
end_line(C)

```
event begin_line(pn_t mpn) {  
    count = 0;  
}
```

```
event call(awsLog_t c) {  
    count++;  
}
```

```
event end_line(pn_t mpn) {  
    msgs m = cm<:mpn:>$msgs;  
    m.out = blendDaily(m.out, count);  
    cm<:mpn:> = m$bins;  
}
```

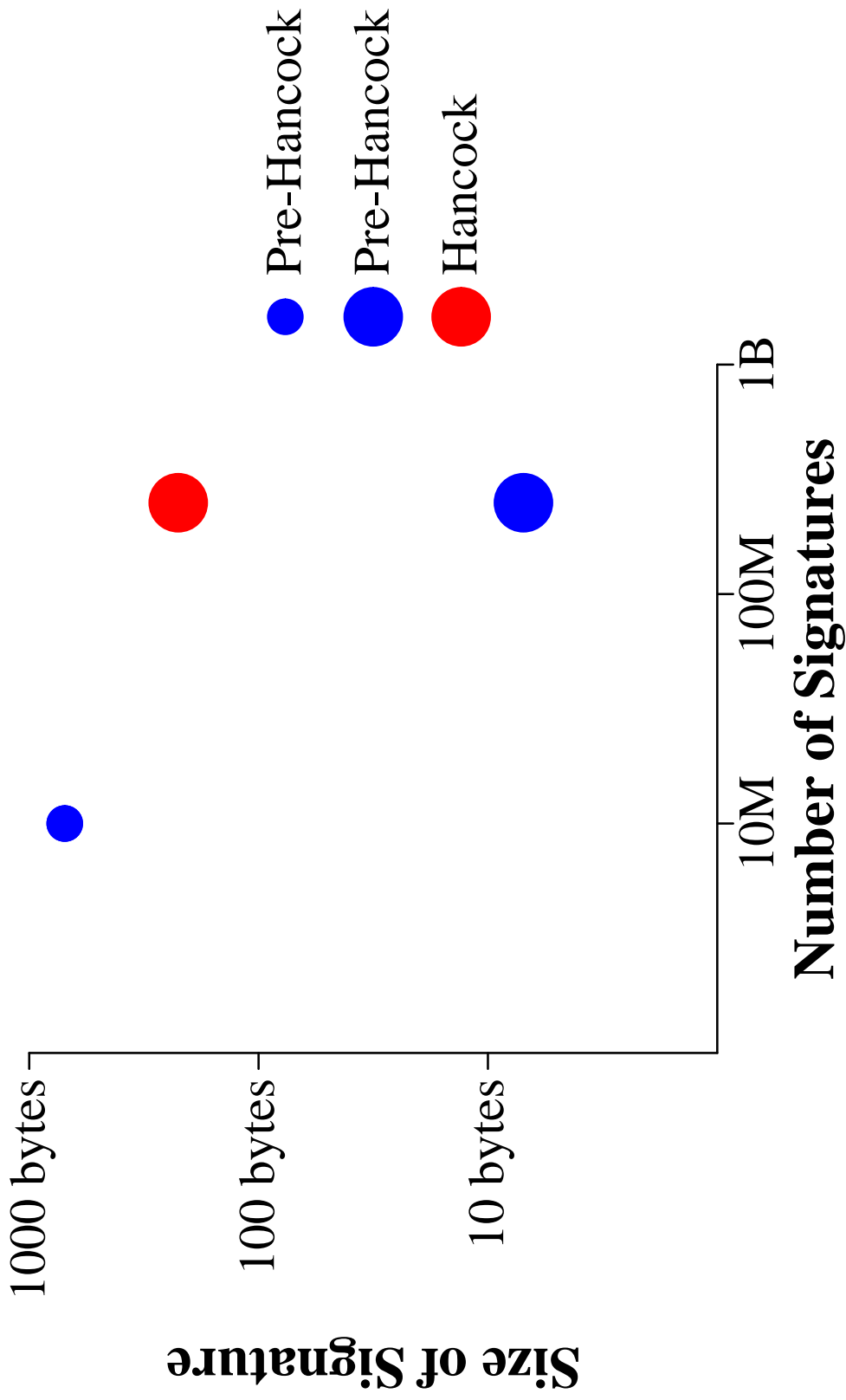
Summary

Hancock:

- Separates **physical** and **logical** representations.
- Captures how signature researchers think about their data.
- Reveals the essence of a computation.

Result is programs that are easy to read, write, and maintain.

Concrete results



Try it!

Hancock compiler/runtime system are available for non-commercial use:

www.research.att.com/projects/hancock

Inquiries to hancock@research.att.com.

Language vs. library

Technical reasons: A library would

- lose Hancock's static typechecking and
- be awkward for expressing Hancock's event model.

Writing a Hancock program is different than writing a C program:

- Hancock removes issues of scale.
- Hancock provides a vocabulary tailored to the application.