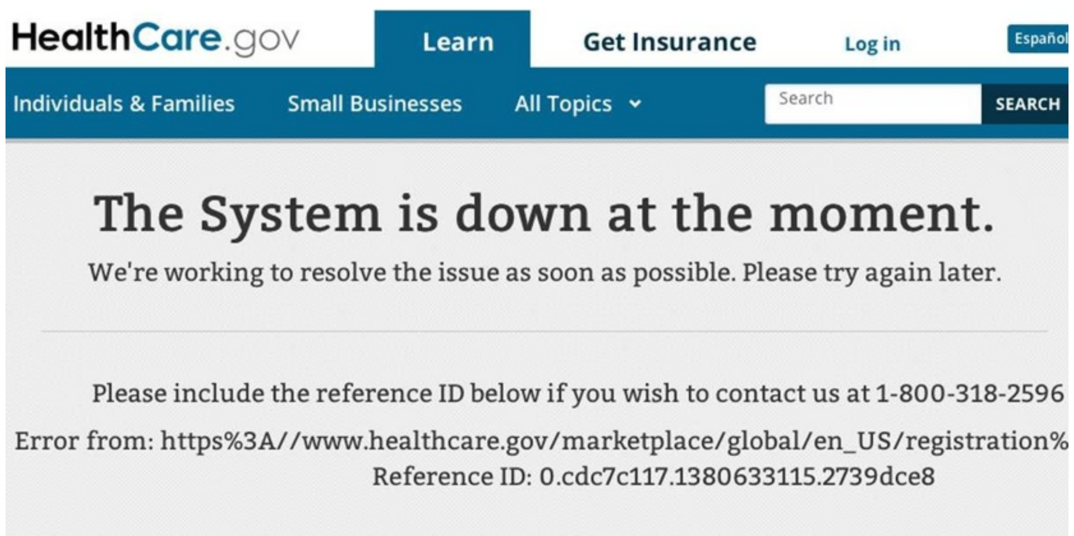
Type-Based Amortized Resource Analysis with Integers and Arrays

Jan Hoffmann and Zhong Shao, Yale University

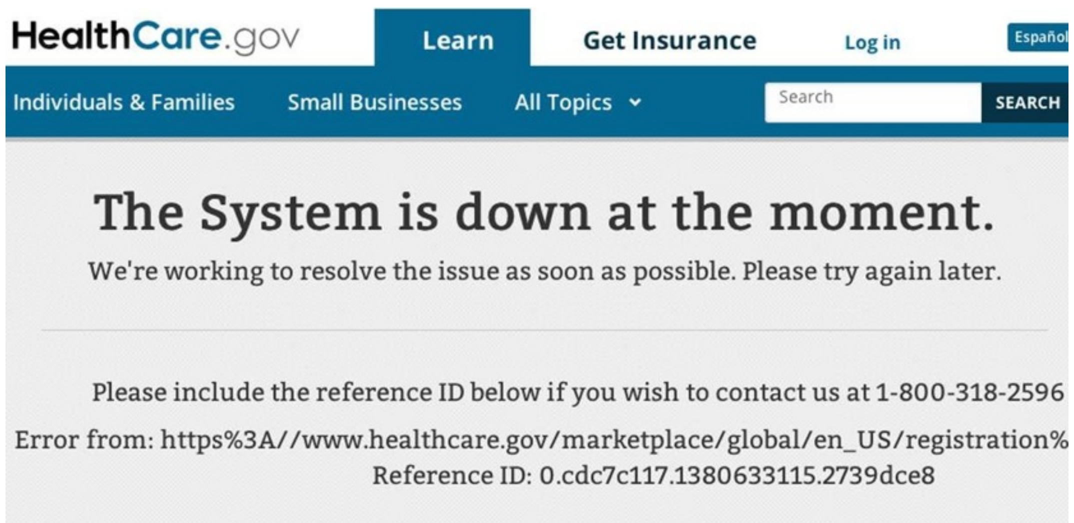
Performance Bugs are Common and Expensive

Performance Bugs are Common and Expensive



HealthCare.gov debacle has been mainly caused by performance issues.

Performance Bugs are Common and Expensive



HealthCare.gov debacle has been mainly caused by performance issues.

ICE 3 Velaro D delivery delayed by one year because of software performance issues in 2013.

Fatalities Because of Stack Overflow?

Fatalities Because of Stack Overflow?

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013
109 Comments

On Thursday October 24, 2013, an Oklahoma court **ruled against Toyota** in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

. . .

Stack overflow. Toyota claimed only 41% of the allocated stack space was being used. Barr's investigation showed that 94% was closer to the truth. On top of that, stack-killing, **MISRA-C** rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against stack overflow.

. . .

Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.

Fatalities Because of Stack Overflow?

EDN NETWORK

About Us · Subscribe to Newsletters

DESIGN CENTERS · TOOLS & LEARNING · COMMUNITY · ED

Home > Automotive Design Center > How To Article

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013

109 Comments

Share 277 · +1 932 · Tweet 724 · Like 3.8k

On Thursday October 24, 2013, an Oklahoma court **ruled against Toyota** in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

. . .

Stack overflow. Toyota claimed only 41% of the allocated stack space was being used. Barr's investigation showed that 94% was closer to the truth. On top of that, stack-killing, **MISRA-C** rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against stack overflow.

. . .

Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.

Expert witness found: "Toyota's electronic throttle control system (ETCS) source code is of unreasonable quality."

Fatalities Because of Stack Overflow?

EDN NETWORK

About Us · Subscribe to Newsletters

DESIGN CENTERS ▾ TOOLS & LEARNING ▾ COMMUNITY ▾ ED

Home > Automotive Design Center > How To Article

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013

109 Comments

Share 277 · +1 932 · Tweet 724 · Like 3.8k

On Thursday October 24, 2013, an Oklahoma court **ruled against Toyota** in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

. . .

Stack overflow. Toyota claimed only 41% of the allocated stack space was being used. Barr's investigation showed that 94% was closer to the truth. On top of that, stack-killing, **MISRA-C** rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against stack overflow.

. . .

Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.

Expert witness found: “Toyota’s electronic throttle control system (ETCS) source code is of unreasonable quality.”

Stack overflow was possible because stack-bound analysis was faulty.

Power Consumption is Increasingly Important

One of the major cost factors in data centers.

Determines battery life in mobile devices and robots.

This Work: Static Resource Analysis

Given: A program P

Question: What is the worst-case resource consumption of P as a function of the size of its inputs?

Static Resource Analysis

- ESOP'10
- APLAS'10
- POPL'11
- PhD Thesis
- CAV'12
- TOPLAS'12
- FLOPS'14

Collaborators: Aehlig (LMU), Hofmann (LMU), Shao (Yale)

This Work: Static Resource Analysis

Clock cycles, heap space, power, ...

Given: A program P

Question: What is the worst-case resource consumption of P as a function of the size of its inputs?

Static Resource Analysis

- ESOP'10
- APLAS'10
- POPL'11
- PhD Thesis
- CAV'12
- TOPLAS'12
- FLOPS'14

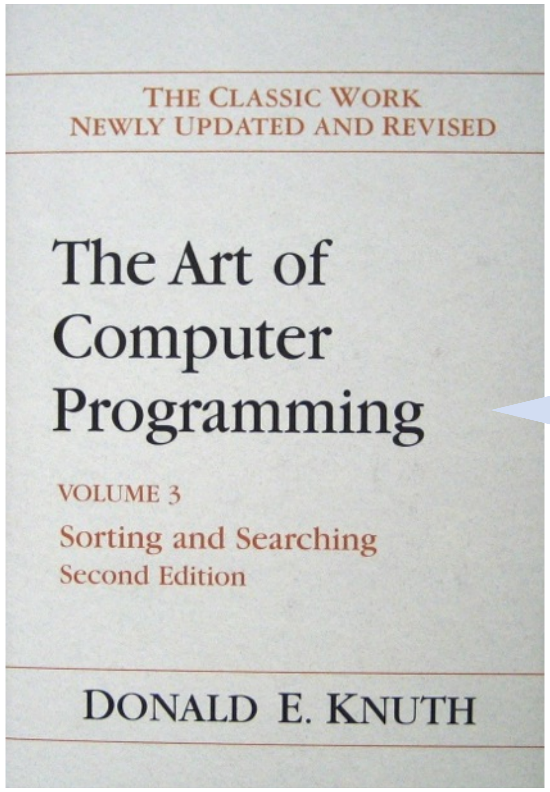
Collaborators: Aehlig (LMU), Hofmann (LMU), Shao (Yale)

This Work: Static Resource Analysis

Given: A program P

Clock cycles, heap space, power, ...

Question: What is the worst-case resource consumption of P as a function of the size of its inputs?



Not only asymptotic bounds but concrete constant factors.

Static Resource Analysis

- ESOP'10
- APLAS'10
- POPL'11
- PhD Thesis
- CAV'12
- TOPLAS'12
- FLOPS'14

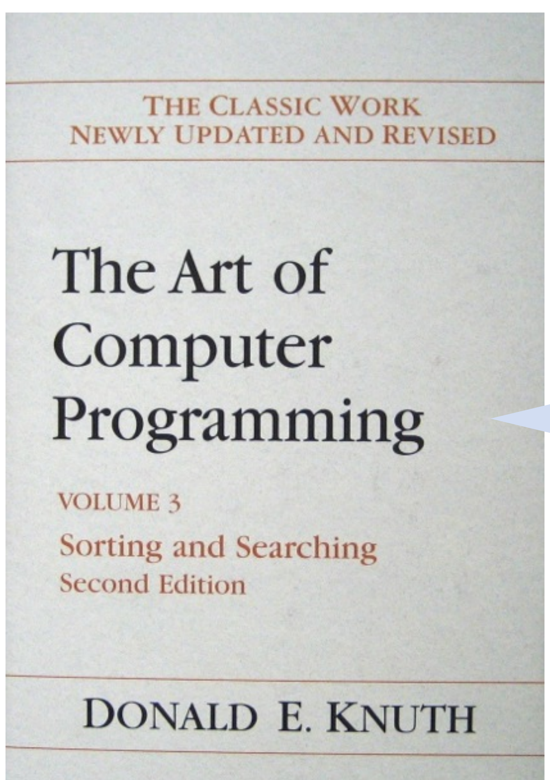
Collaborators: Aehlig (LMU), Hofmann (LMU), Shao (Yale)

This Work: Automatic Static Resource Analysis

Clock cycles, heap space, power, ...

Given: A program P

Question: What is the worst-case resource consumption of P as a function of the size of its inputs?



Not only asymptotic bounds but concrete constant factors.

Static Resource Analysis

- ESOP'10
- APLAS'10
- POPL'11
- PhD Thesis
- CAV'12
- TOPLAS'12
- FLOPS'14

Collaborators: Aehlig (LMU), Hofmann (LMU), Shao (Yale)

Our Approach: Type-Based Resource Analysis

- Start with a *functional* programming language

1. **Model** the resource usage of programs with an operational semantics

Our Approach: Type-Based Resource Analysis

- Start with a *functional* programming language

1. **Model** the resource usage of programs with an operational semantics

2. Define a **type system** so that type derivations establish resource bounds

Our Approach: Type-Based Resource Analysis

- Start with a *functional* programming language

1. **Model** the resource usage of programs with an operational semantics

2. Define a **type system** so that type derivations establish resource bounds

3. Prove the **soundness** of the types system with respect to the semantics

Our Approach: Type-Based Resource Analysis

- Start with a *functional* programming language

1. **Model** the resource usage of programs with an operational semantics
2. Define a **type system** so that type derivations establish resource bounds
3. Prove the **soundness** of the types system with respect to the semantics
4. Develop an **efficient inference** algorithm for the type system

Our Approach: Type-Based Resource Analysis

- Start with a *functional* programming language

1. **Model** the resource usage of programs with an operational semantics

2. Define a **type system** so that type derivations establish resource bounds

3. Prove the **soundness** of the types system with respect to the semantics

   Undecidable!

4. Develop an **efficient inference** algorithm for the type system

Our Approach: Type-Based Resource Analysis

- Start with a *functional* programming language

1. **Model** the resource usage of programs with an operational semantics

2. Define a **type system** so that type derivations establish resource bounds

3. Prove the **soundness** of the types system with respect to the semantics

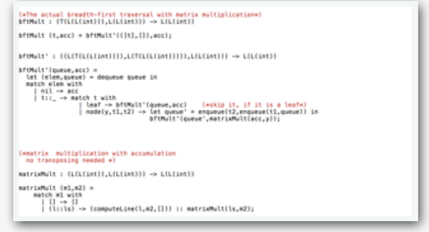
   Undecidable!

4. Develop an **efficient inference** algorithm for the type system

5. Show the practicality of the system with an **implementation and experiments**

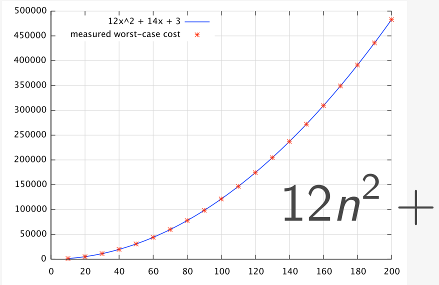
Polynomial Amortized Resource Analysis

- Automatic type-based analysis: **No annotations required**
- **Naturally compositional:** function types are resource specifications
- **Generic in the resource:** heap space, clock cycles, energy usage ...
- Precise bounds expressed by **multivariate resource polynomials**
- Efficient type inference based on **linear programming**



```
(*The actual breadth-first traversal with matrix multiplication*)
bftMult : (T(L(L(int))),L(L(int))) -> L(L(int))

bftMult (t,acc) = bftMult'(([t],[]),acc);


bftMult' : ((L(T(L(L(int)))),L(T(L(L(int))))),L(L(int))) -> L(L(int))

bftMult'(queue,acc) =
  let (elem,queue) = dequeue queue in
  match elem with
    | nil -> acc
    | t::_ -> match t with
                | leaf -> bftMult'(queue,acc)    (*skip it, if it is a leaf*)
                | node(y,t1,t2) -> let queue' = enqueue(t2,enqueue(t1,queue)) in
                                   bftMult'(queue',matrixMult(acc,y));



(*matrix  multiplication with accumulation
  no transposing needed *)

matrixMult : (L(L(int)),L(L(int))) -> L(L(int))

matrixMult (m1,m2) =
    match m1 with
      | [] -> []
      | (l::ls) -> (computeLine(l,m2,[])) :: matrixMult(ls,m2);
```

Source Code



$12n^2 + 14n + 3$

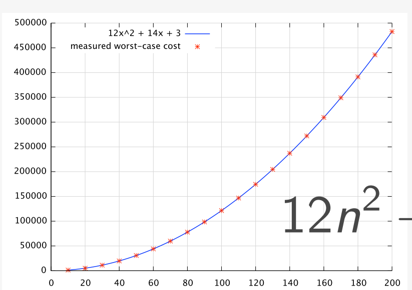
Resource Bound

Bird’s Eye View

Type-Based Resource Analysis

```
(*The actual breadth-first traversal with matrix multiplication*)
bftMult : (T(L(L(int))),L(L(int))) -> L(L(int))

bftMult (t,acc) = bftMult'(([t],[]),acc);


bftMult' : (((L(T(L(L(int)))),L(T(L(L(int))))),L(L(int))) -> L(L(int))

bftMult'(queue,acc) =
  let (elem,queue) = dequeue queue in
  match elem with
    | nil -> acc
    | t::_ -> match t with
                | leaf -> bftMult'(queue,acc)    (*skip it, if it is a leaf*)
                | node(y,t1,t2) -> let queue' = enqueue(t2,enqueue(t1,queue)) in
                                   bftMult'(queue',matrixMult(acc,y));



(*matrix  multiplication with accumulation
  no transposing needed *)

matrixMult : (L(L(int)),L(L(int))) -> L(L(int))

matrixMult (m1,m2) =
    match m1 with
      | [] -> []
      | (l::ls) -> (computeLine(l,m2,[])) :: matrixMult(ls,m2);
```

Source Code



$12n^2 + 14n + 3$

Resource Bound

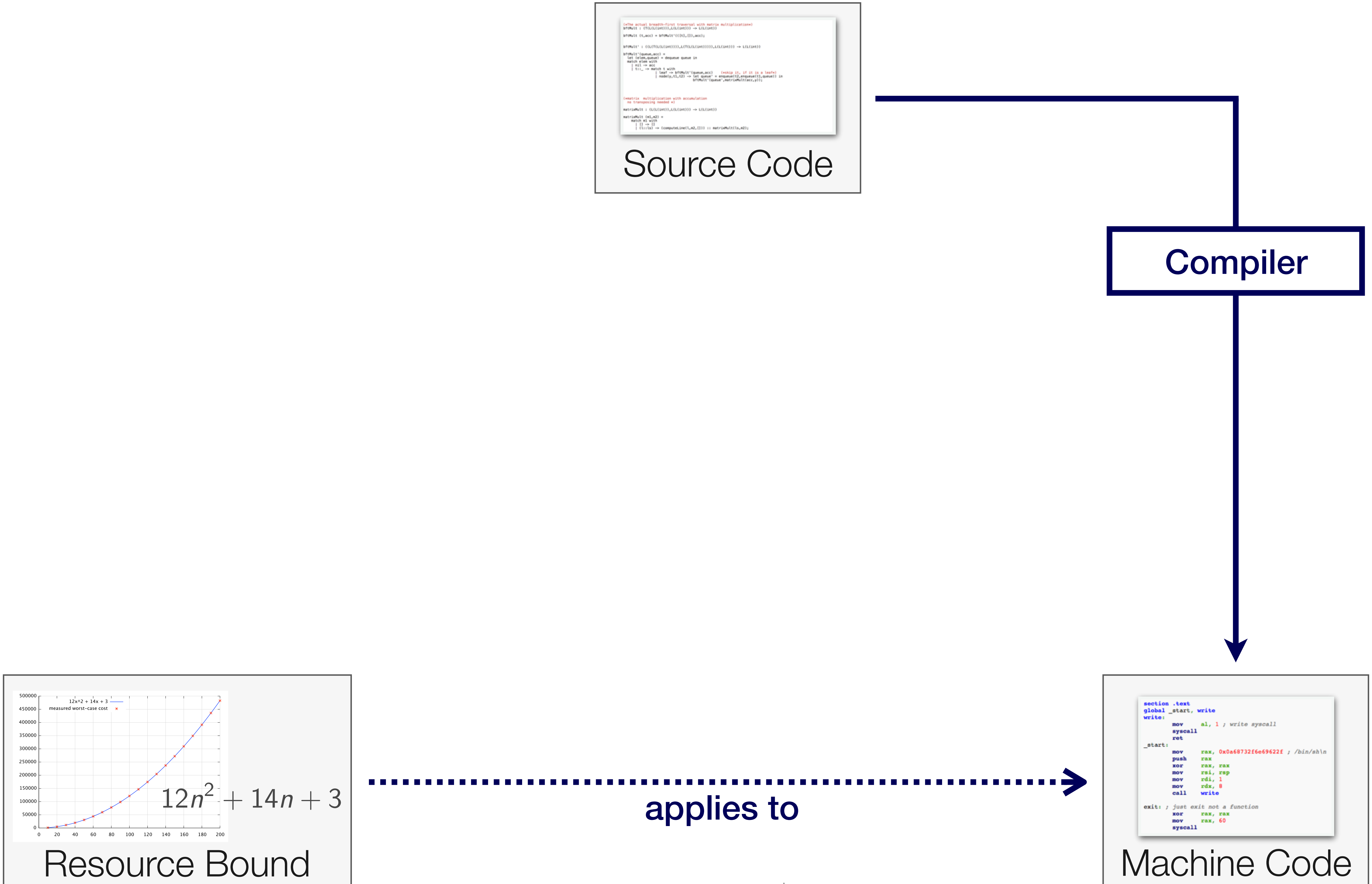
applies to

```
section .text
global _start, write
write:
        mov     al, 1 ; write syscall
        syscall
        ret
_start:
        mov     rax, 0x0a68732f6e69622f ; /bin/sh\n
        push    rax
        xor     rax, rax
        mov     rsi, rsp
        mov     rdi, 1
        mov     rdx, 8
        call    write

exit: ; just exit not a function
        xor     rax, rax
        mov     rax, 60
        syscall
```

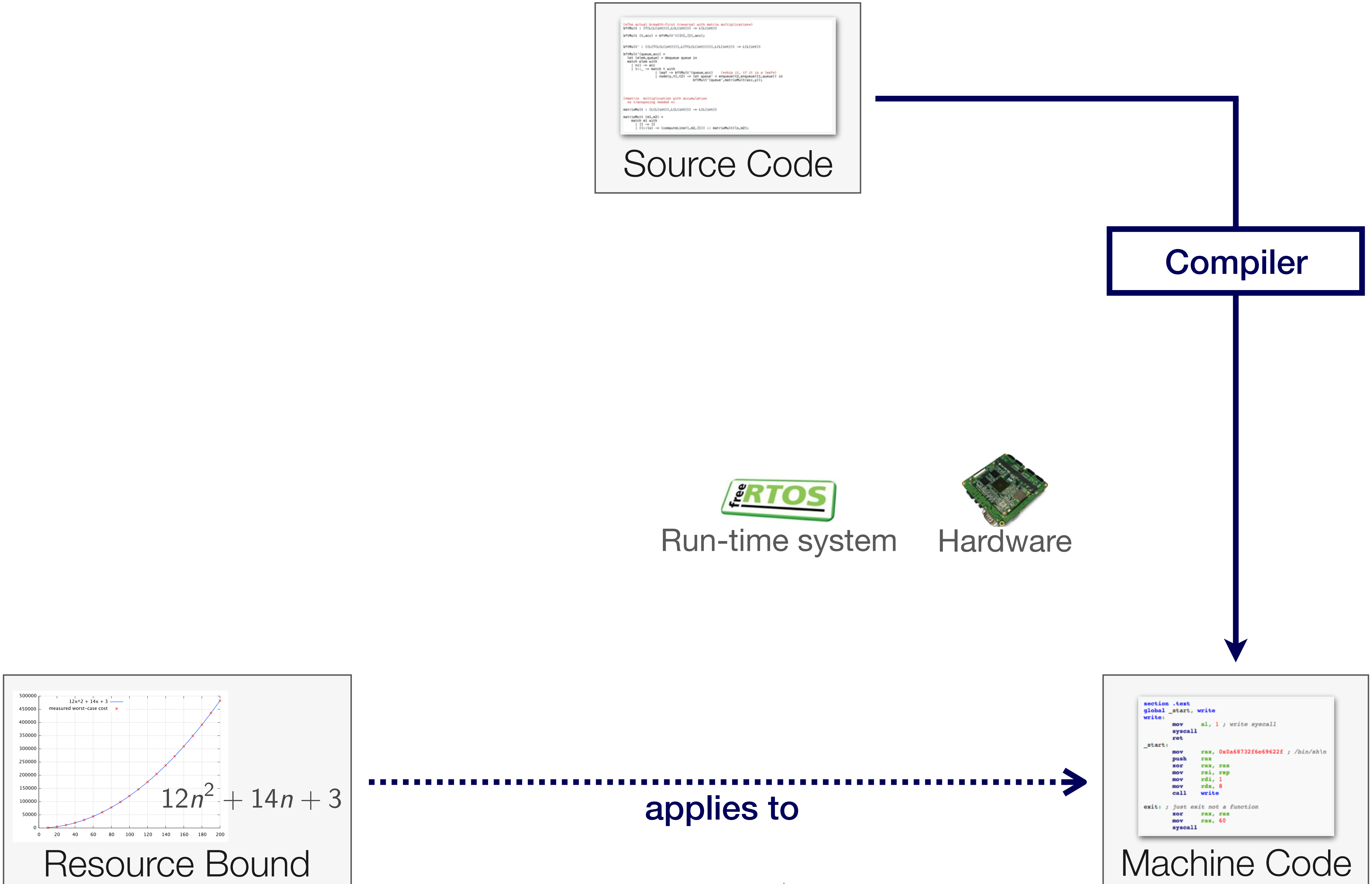
Machine Code

Bird’s Eye View

Type-Based Resource Analysis

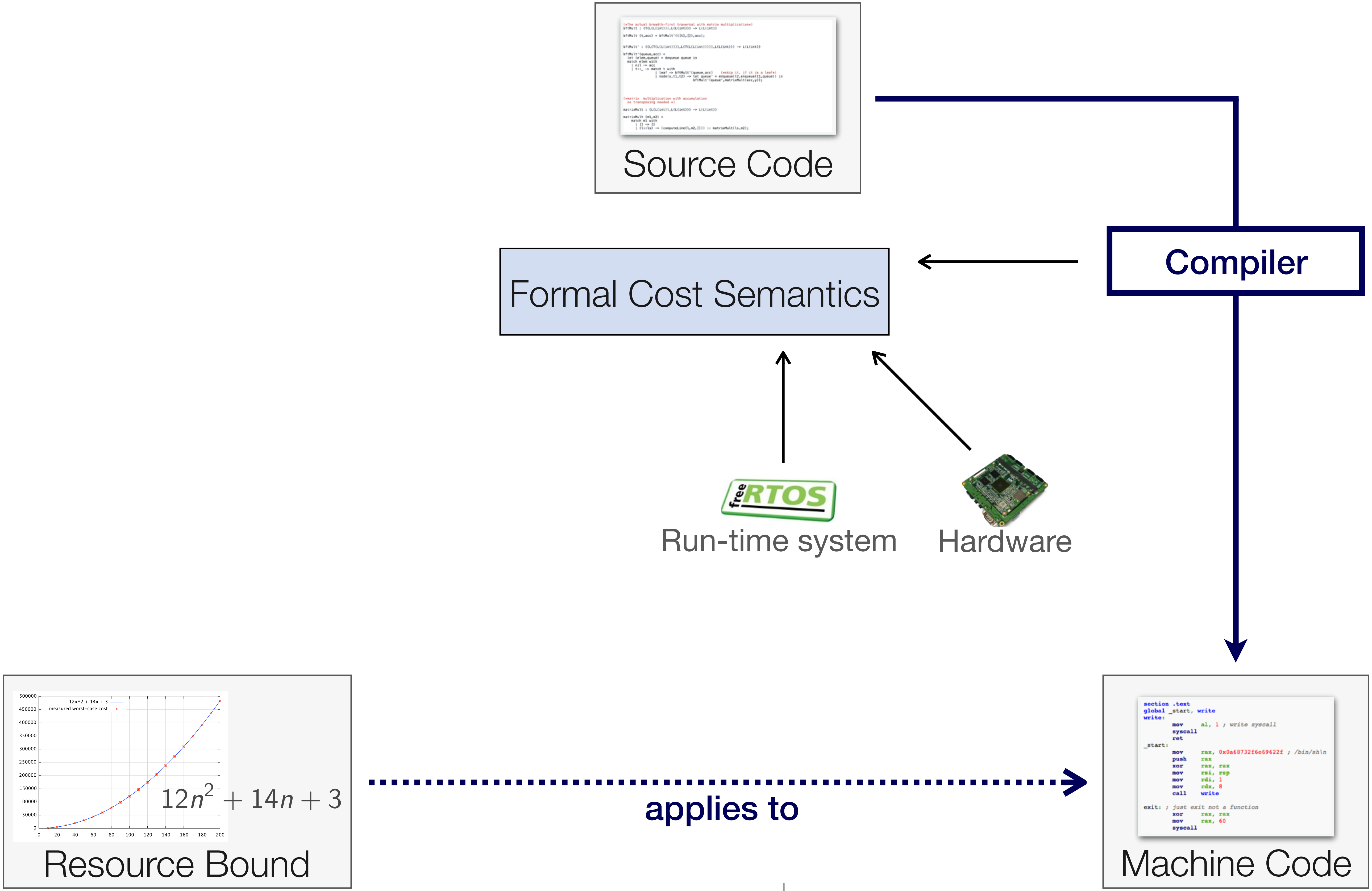


Bird’s Eye View



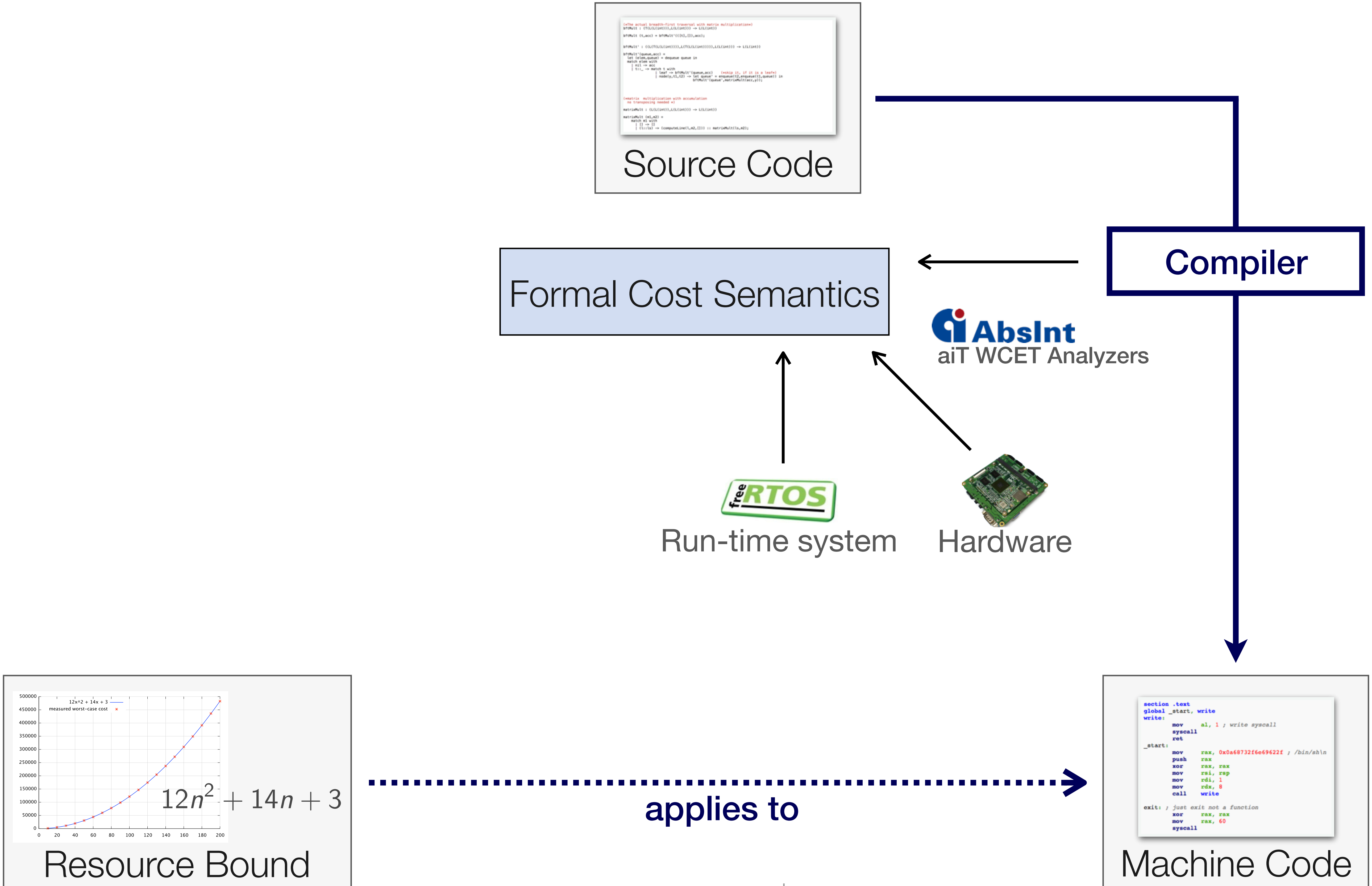
Bird’s Eye View

Type-Based Resource Analysis



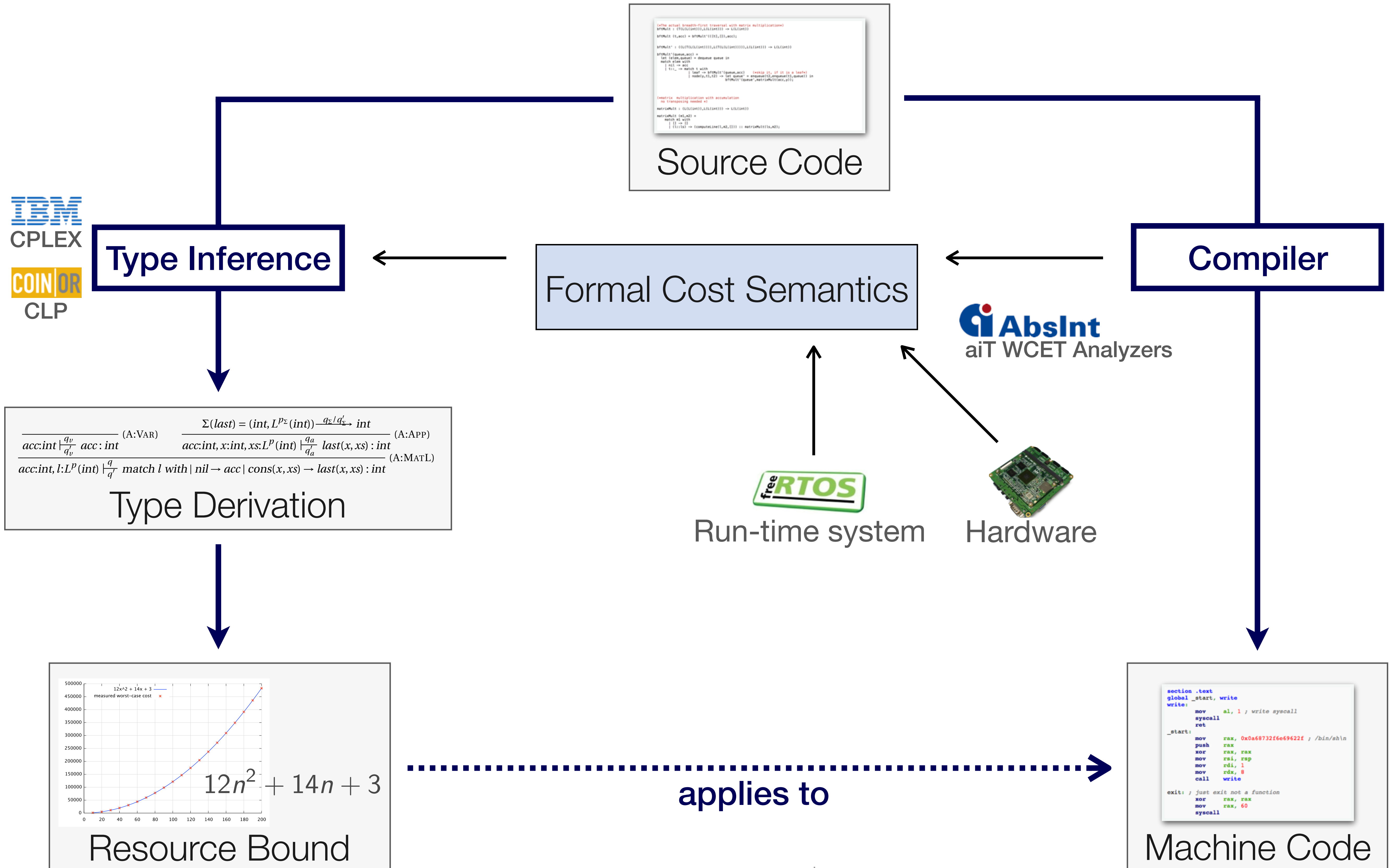
Bird's Eye View

Type-Based Resource Analysis



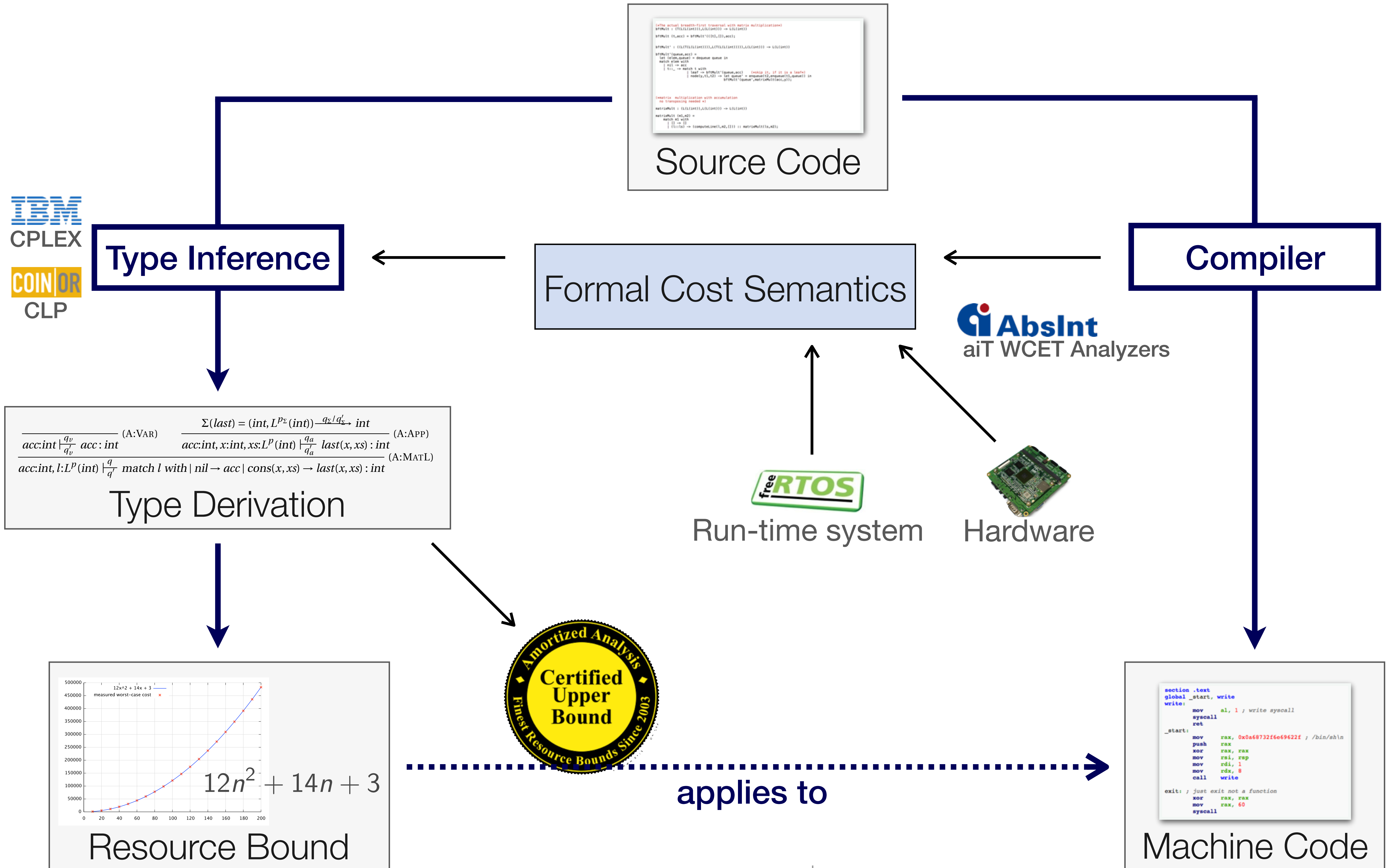
Bird’s Eye View

Type-Based Resource Analysis



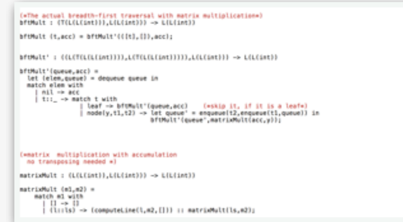
Bird's Eye View

Type-Based Resource Analysis



Bird's Eye View

Type-Based Resource Analysis



Source Code

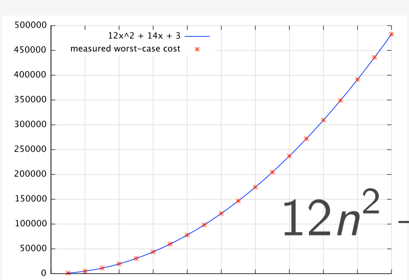
IBM CPLEX

COIN|OR CLP

Type Inference

$$\dfrac{\dfrac{}{acc{:}int \vdash^{q_v}_{q'_v} acc : int}\ \text{(A:VAR)} \qquad \dfrac{\Sigma(last) = (int, L^{p_\Sigma}(int)) \xrightarrow{q_\Sigma / q'_\Sigma} int}{acc{:}int, x{:}int, xs{:}L^p(int) \vdash^{q_a}_{q'_a} last(x,xs) : int}\ \text{(A:APP)}}{acc{:}int, l{:}L^p(int) \vdash^{q}_{q'} \ match\ l\ with \mid nil \to acc \mid cons(x,xs) \to last(x,xs) : int}\ \text{(A:MATL)}$$

Type Derivation



$12n^2 + 14n + 3$

Resource Bound

Bird’s Eye View

Type-Based Resource Analysis

Bird's Eye View

Type-Based Resource Analysis

Source Code

Type Inference

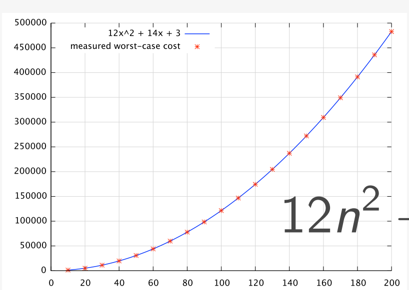
IBM CPLEX

COIN|OR CLP

$$\dfrac{\dfrac{}{acc{:}int \mid_{q_v'}^{q_v} acc : int}\,(\text{A:Var}) \qquad \dfrac{\Sigma(last) = (int, L^{p_\Sigma}(int)) \xrightarrow{q_\Sigma / q_\Sigma'} int}{acc{:}int, x{:}int, xs{:}L^{p}(int) \mid_{q_a'}^{q_a} last(x,xs) : int}\,(\text{A:App})}{acc{:}int, l{:}L^{p}(int) \mid_{q'}^{q} match\ l\ with \mid nil \to acc \mid cons(x,xs) \to last(x,xs) : int}\,(\text{A:MatL})$$

Type Derivation

Polynomial Amortized Resource Analysis



$12n^2 + 14n + 3$

Resource Bound

Can we transfer the ideas of automatic amortized analysis to C-like programs?

Why Automatic Amortized Analysis for C Code?

- Today's embedded and real-time systems are written in C code

- There are many great techniques for deriving resource bounds on imperative code [Gulwani et al., Albert et al., Brockschmidt et al.]

  But: current techniques are not compositional

Why looking a functional programs in the first place?

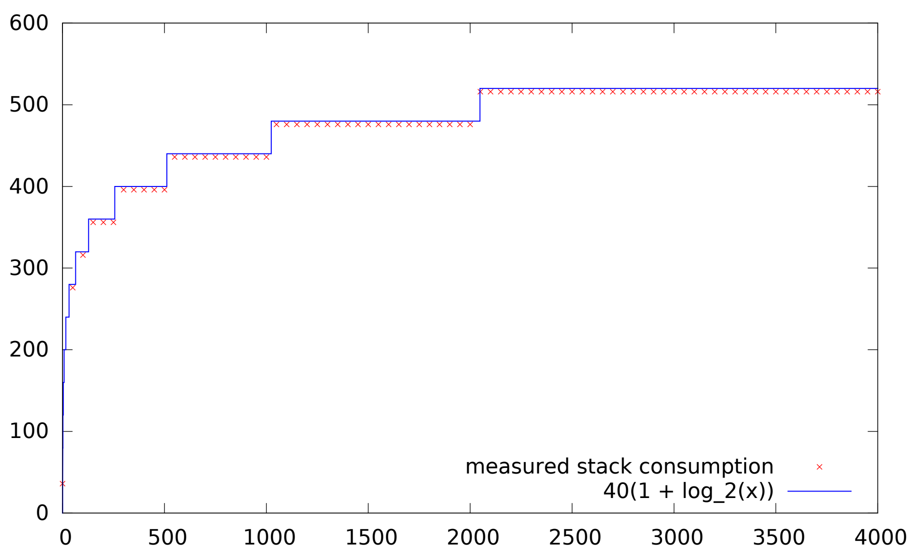
- Might be used more often in the future

- Clean setting to study and understand the problem (compare: type systems, type inference, higher-order functions, …)

Promising First Results: Stack Bounds

[PLDI’14]: End-to-End Verification of Stack-Space Bounds for C Programs

- Uses CompCert and a program logic that is based on amortized analysis
- Verified in Coq
- Applied to the CertiKOS hypervisor kernel

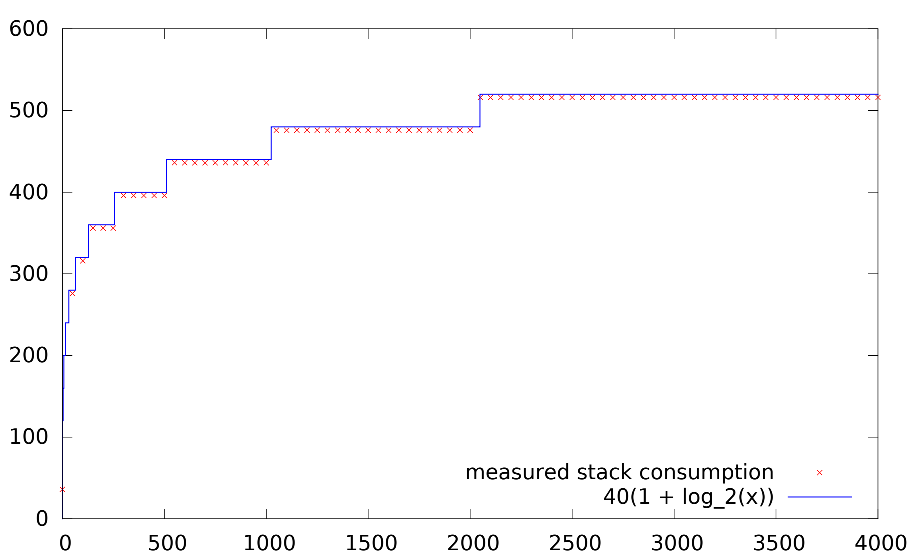


Function Name	Verified Stack Bound
$\mathsf{recid}()$	$8\mathsf{a}$ bytes
$\mathsf{bsearch}(\mathsf{x}, \mathsf{lo}, \mathsf{hi})$	$40(1 + \log_2(\mathsf{hi} - \mathsf{lo}))$ bytes
$\mathsf{fib}(\mathsf{n})$	$24\mathsf{n}$ bytes
$\mathsf{qsort}(\mathsf{a}, \mathsf{lo}, \mathsf{hi})$	$48(\mathsf{hi} - \mathsf{lo})$ bytes
$\mathsf{filter\_pos}(\mathsf{a}, \mathsf{sz}, \mathsf{lo}, \mathsf{hi})$	$48(\mathsf{hi} - \mathsf{lo})$ bytes
$\mathsf{sum}(\mathsf{a}, \mathsf{lo}, \mathsf{hi})$	$32(\mathsf{hi} - \mathsf{lo})$ bytes
$\mathsf{fact\_sq}(\mathsf{n})$	$40 + 24\mathsf{n}^2$ bytes
$\mathsf{filter\_find}(\mathsf{a}, \mathsf{sz}, \mathsf{lo}, \mathsf{hi})$	$128 + 48(\mathsf{hi} - \mathsf{lo}) + 40\log_2(\mathsf{BL})$ bytes

Promising First Results: Stack Bounds

[PLDI'14]: End-to-End Verification of Stack-Space Bounds for C Programs

- Uses CompCert and a program logic that is based on amortized analysis
- Verified in Coq
- Applied to the CertiKOS hypervisor kernel



Function Name	Verified Stack Bound
recid()	$8a$ bytes
bsearch(x, lo, hi)	$40(1 + \log_2(\mathsf{hi} - \mathsf{lo}))$ bytes
fib(n)	$24n$ bytes
qsort(a, lo, hi)	$48(\mathsf{hi} - \mathsf{lo})$ bytes
filter_pos(a, sz, lo, hi)	$48(\mathsf{hi} - \mathsf{lo})$ bytes
sum(a, lo, hi)	$32(\mathsf{hi} - \mathsf{lo})$ bytes
fact_sq(n)	$40 + 24n^2$ bytes
filter_find(a, sz, lo, hi)	$128 + 48(\mathsf{hi} - \mathsf{lo}) + 40\log_2(\mathsf{BL})$ bytes

Promising First Results: Stack Bounds

[PLDI'14]: End-to-End Verification of Stack-Space Bounds for C Programs

- Uses CompCert and a program logic that is based on amortized analysis
- Verified in Coq
- Applied to the CertiKOS hypervisor kernel


EDN NETWORK
About Us · Subscribe to Newsletters
DESIGN CENTERS TOOLS & LEA

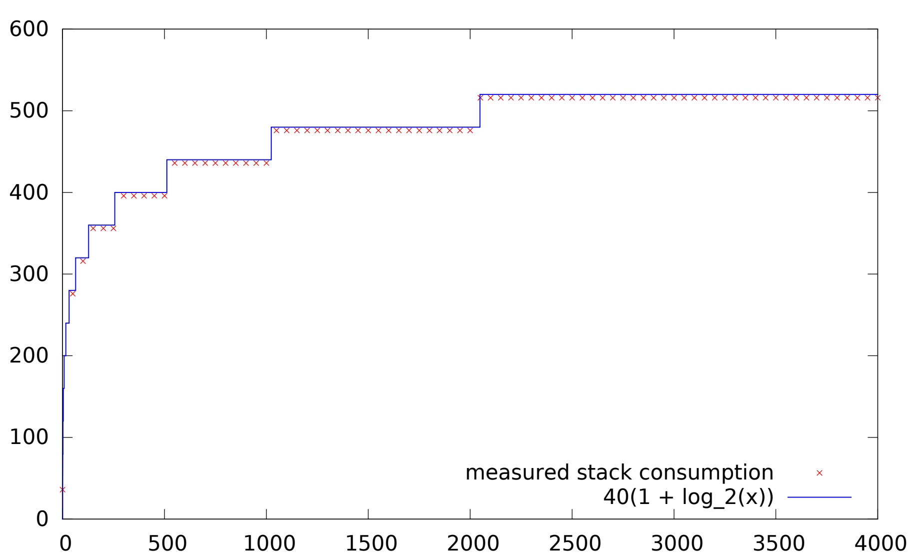
Home > Automotive Design Center > How To Article

Toyota's killer firmware: Bad
its consequences

Michael Dunn -October 28, 2013
109 Comments

Share 277 · g+1 932 · Tweet 724 · Like 3.8k

On Thursday October 24, 2013, an Oklahoma court ruled against Toy
acceleration that lead to the death of one the occupants. Central to
Control Module's (ECM) firmware.

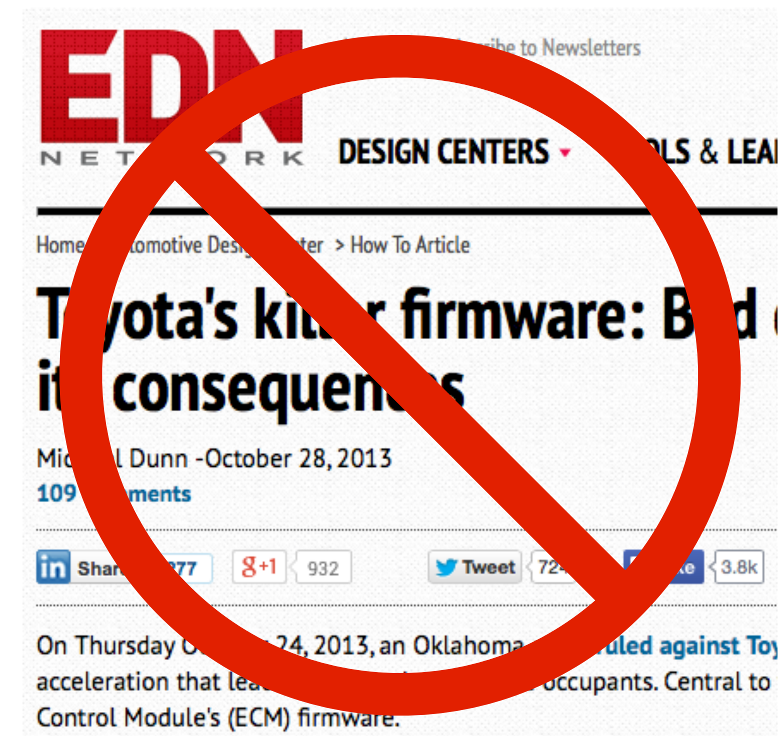


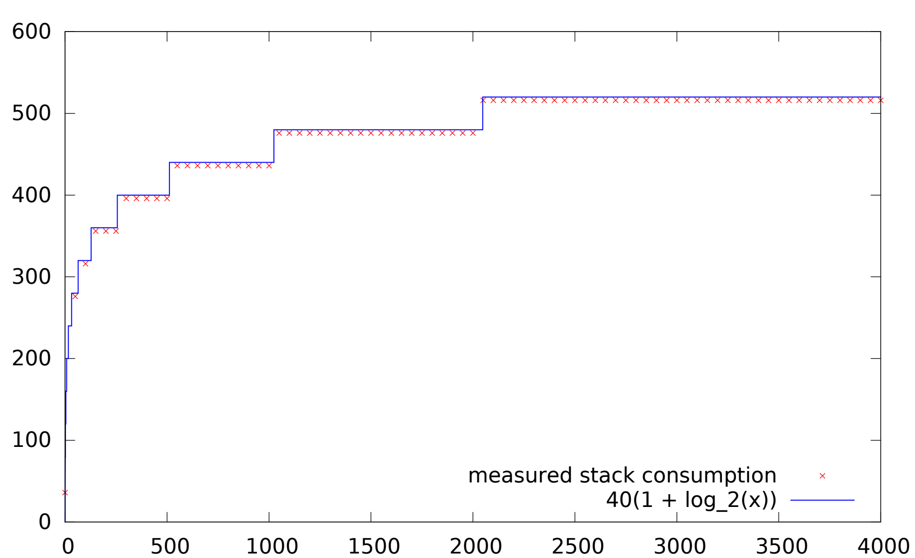
Function Name	Verified Stack Bound
recid()	8a bytes
bsearch(x, lo, hi)	$40(1 + \log_2(\mathrm{hi} - \mathrm{lo}))$ bytes
fib(n)	24n bytes
qsort(a, lo, hi)	$48(\mathrm{hi} - \mathrm{lo})$ bytes
filter_pos(a, sz, lo, hi)	$48(\mathrm{hi} - \mathrm{lo})$ bytes
sum(a, lo, hi)	$32(\mathrm{hi} - \mathrm{lo})$ bytes
fact_sq(n)	$40 + 24n^2$ bytes
filter_find(a, sz, lo, hi)	$128 + 48(\mathrm{hi} - \mathrm{lo}) + 40\log_2(\mathrm{BL})$ bytes

Promising First Results: Stack Bounds

[PLDI'14]: End-to-End Verification of Stack-Space Bounds for C Programs

- Uses CompCert and a program logic that is based on amortized analysis
- Verified in Coq
- Applied to the CertiKOS hypervisor kernel





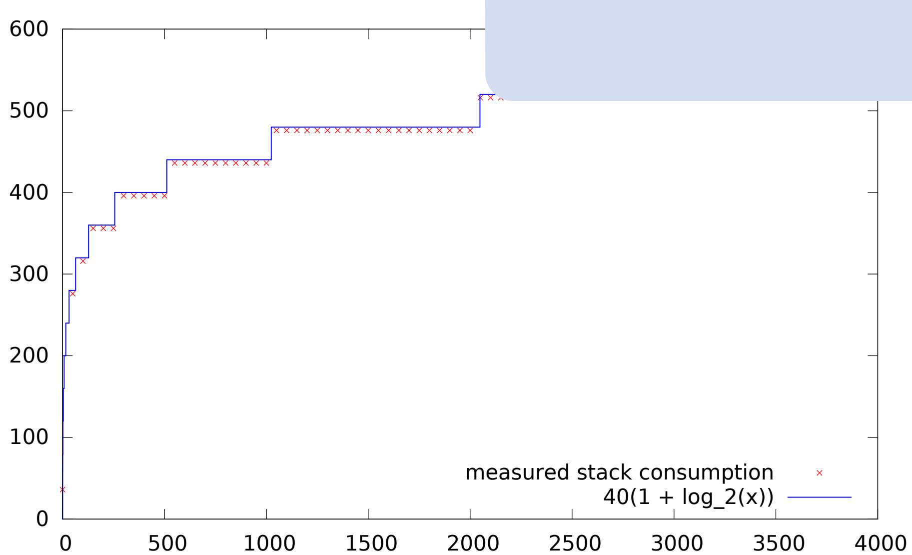
Function Name	Verified Stack Bound
recid()	$8a$ bytes
bsearch(x, lo, hi)	$40(1 + \log_2(\mathrm{hi} - \mathrm{lo}))$ bytes
fib(n)	$24n$ bytes
qsort(a, lo, hi)	$48(\mathrm{hi} - \mathrm{lo})$ bytes
filter_pos(a, sz, lo, hi)	$48(\mathrm{hi} - \mathrm{lo})$ bytes
sum(a, lo, hi)	$32(\mathrm{hi} - \mathrm{lo})$ bytes
fact_sq(n)	$40 + 24n^2$ bytes
filter_find(a, sz, lo, hi)	$128 + 48(\mathrm{hi} - \mathrm{lo}) + 40\log_2(\mathrm{BL})$ bytes

Promising First Results: Stack Bounds

[PLDI'14]: End-to-End Verification of Stack-Space Bounds for C Programs

- Uses CompCert and a program logic that is based on amortized analysis
- Verified in (
- Applied to

Automation only for programs without recursion!



	ck Bound
recid()	8a bytes
bsearch(x, lo, hi)	$40(1 + \log_2(\mathsf{hi} - \mathsf{lo}))$ bytes
fib(n)	24n bytes
qsort(a, lo, hi)	$48(\mathsf{hi} - \mathsf{lo})$ bytes
filter_pos(a, sz, lo, hi)	$48(\mathsf{hi} - \mathsf{lo})$ bytes
sum(a, lo, hi)	$32(\mathsf{hi} - \mathsf{lo})$ bytes
fact_sq(n)	$40 + 24n^2$ bytes
filter_find(a, sz, lo, hi)	$128 + 48(\mathsf{hi} - \mathsf{lo}) + 40 \log_2(\mathsf{BL})$ bytes

Challenges in (Numeric) Imperative code?

Challenges in (Numeric) Imperative code?

	Functional	Imperative
Data structures	Inductive data types	Arrays
Iteration	Recursion	Loops
Control Flow	Pattern matching	Integers

Challenges in (Numeric) Imperative code?

	Functional	Imperative
Data structures	Inductive data types	Arrays
Iteration	Recursion	Loops
Control Flow	Pattern matching	Integers

Cost depends on	Sizes of ind. data structures	Sizes of integer intervals |[n,m]|
Size changes in	Pattern matching, constructors	Arithmetic operations

Challenges in (Numeric) Imperative code?

	Functional	Imperative
Data structures	Inductive data types	Arrays
Iteration	Recursion	Loops
Control Flow	Pattern matching	Integers

Cost depends on	Sizes of ind. data structures	Sizes of integer intervals |[n,m]|
Size changes in	Pattern matching, constructors	Arithmetic operations

1. **Track size changes in arithmetic operations**

2. **Apply the analysis to C programs and track sizes of intervals**

Challenges in (Numeric) Imperative code?

	Functional	Imperative
Data structures	Inductive data types	Arrays
Iteration	Recursion	Loops
Control Flow	Pattern matching	Integers

Cost depends on	Sizes of ind. data structures	Sizes of integer intervals |[n,m]|
Size changes in	Pattern matching, constructors	Arithmetic operations

1. Track size changes in arithmetic operations

Today.

2. Apply the analysis to C programs and track sizes of intervals

Challenges in (Numeric) Imperative code?

	Functional	Imperative
Data structures	Inductive data types	Arrays
Iteration	Recursion	Loops
Control Flow	Pattern matching	Integers

Cost depends on	Sizes of ind. data structures	Sizes of integer intervals |[n,m]|
Size changes in	Pattern matching, constructors	Arithmetic operations

1. Track size changes in arithmetic operations

Today.

2. Apply the analysis to C programs and track sizes of intervals

Upcoming paper.

The General Idea of Amortized Analysis

- Assign potential functions to data structures
  ➡ States are mapped to non-negative numbers

$$\Phi(\mathit{state}) \geq 0$$

- Potential pays the resource consumption and the potential at the following program point

$$\Phi(\mathit{before}) \geq \Phi(\mathit{after}) + \mathit{cost}$$

$\Downarrow$ telescoping $\Downarrow$

- Initial potential is an upper bound

$$\Phi(\mathit{initial\ state}) \geq \sum \mathit{cost}$$

The General Idea of Amortized Analysis

- Assign potential functions to data structures
  - ➡ States are mapped to non-negative numbers

$$\Phi(\mathit{state}) \geq 0$$

- Potential pays the resource consumption and the potential at the following program point

$$\Phi(\mathit{before}) \geq \Phi(\mathit{after}) + \mathit{cost}$$

↓ telescoping ↓

- Initial potential is an upper bound

$$\Phi(\mathit{initial\ state}) \geq \sum \mathit{cost}$$

Type Systems for automatic analysis

- Fix a format of potential functions
- Develop type rules that manipulate potential functions

The General Idea of Amortized Analysis

- Assign potential functions to data structures
  - ➡ States are mapped to non-negative numbers

$$\Phi(\mathit{state}) \geq 0$$

- Potential pays the resource consumption and the potential at the following program point

$$\Phi(\mathit{before}) \geq \Phi(\mathit{after}) + \mathit{cost}$$

$$\Big\downarrow \text{telescoping} \Big\downarrow$$

- Initial potential is an upper bound

$$\Phi(\mathit{initial\ \ state}) \geq \sum \mathit{cost}$$

Type Systems for automatic analysis

Potential is given by type context.

- Fix a format of potential functions
- Develop type rules that manipulate potential functions

Programs with Unsigned Integers (nat)

Data types:	$(\mathsf{nat} * \mathsf{nat}, (q_{(i,j)})_{i,j\in\mathbb{N}})$
Potential functions:	$\Phi((n,m),(q_{(i,j)})) = \sum_{i,j\in\mathbb{N}} q_{(i,j)} \binom{n}{i}\binom{m}{j}$

Function types: $(A, Q) \to (B, Q')$

Programs with Unsigned Integers (nat)

Non-negative rational numbers

Data types:	$(\mathtt{nat} * \mathtt{nat}, (q_{(i,j)})_{i,j\in\mathbb{N}})$
Potential functions:	$\Phi((n,m),(q_{(i,j)})) = \sum_{i,j\in\mathbb{N}} q_{(i,j)} \binom{n}{i}\binom{m}{j}$

Function types: $(A, Q) \to (B, Q')$

Programs with Unsigned Integers (nat)

Non-negative rational numbers

Data types:	$(\mathsf{nat} * \mathsf{nat}, (q_{(i,j)})_{i,j \in \mathbb{N}})$
Potential functions:	$\Phi((n,m),(q_{(i,j)})) = \sum_{i,j \in \mathbb{N}} q_{(i,j)} \binom{n}{i} \binom{m}{j}$

Function types: $(A, Q) \rightarrow (B, Q')$

Polynomial input potential

Polynomial output potential

Example: Evaluation Steps of `mult`

```
add : (nat,nat) -> nat

add(n,m) =
 match n with | 0 -> m
  | n+1 -> 1+add(n,m);
```

```
mult: (nat,nat) -> nat

mult(n,m) =
 match n with | 0 -> 0
  | n+1 -> add(m,mult(n,m));
```

Example: Evaluation Steps of `mult`

```
add : (nat,nat) -> nat

add(n,m) =
 match n with | 0 -> m
  | n+1 -> 1+add(n,m);
```

```
mult: (nat,nat) -> nat

mult(n,m) =
 match n with | 0 -> 0
  | n+1 -> add(m,mult(n,m));
```

Number of evaluation steps of `mult` in the worst case: $8nm + 12n + 3$

Example: Evaluation Steps of `mult`

```
add : (nat,nat) -> nat

add(n,m) =
 match n with | 0 -> m
  | n+1 -> 1+add(n,m);
```

```
mult: (nat,nat) -> nat

mult(n,m) =
 match n with | 0 -> 0
  | n+1 -> add(m,mult(n,m));
```

Number of evaluation steps of `mult` in the worst case: $8nm + 12n + 3$

Possible typing of `mult:` $(\mathsf{nat} * \mathsf{nat}, (q_{(i,j)})_{i,j \in \mathbb{N}}) \to (\mathsf{nat}, (p_i)_{i \in \mathbb{N}})$

where

$$
\begin{aligned}
q_{(0,0)} &= 3 \\
q_{(1,0)} &= 12 \\
q_{(1,1)} &= 8 \\
q_{(i,j)} &= 0 \text{ otherwise}
\end{aligned}
$$

$$p_i = 0 \text{ for all } i$$

Example: Evaluation Steps of `mult`

```
add : (nat,nat) -> nat

add(n,m) =
 match n with | 0 -> m
  | n+1 -> 1+a
```

```
mult: (nat,nat) -> nat

mult(n,m) =
 match n with | 0 -> 0
  | n+1 -> add(m,mult(n,m));
```

$$\Phi((n,m),(q_{(i,j)})) = \sum_{i,j\in\mathbb{N}} q_{(i,j)} \binom{n}{i}\binom{m}{j}$$

Number of evalua he worst case: $8nm + 12n + 3$

Possible typing of `mult:` $(\mathsf{nat} * \mathsf{nat}, (q_{(i,j)})_{i,j\in\mathbb{N}}) \to (\mathsf{nat}, (p_i)_{i\in\mathbb{N}})$

where

$$\begin{aligned} q_{(0,0)} &= 3 \\ q_{(1,0)} &= 12 \\ q_{(1,1)} &= 8 \\ q_{(i,j)} &= 0 \text{ otherwise} \end{aligned}$$

$p_i = 0$ for all i

Example: Evaluation Steps of `mult`

```
add : (nat,nat) -> nat

add(n,m) =
 match n with | 0 -> m
  | n+1 -> 1+a
```

```
mult: (nat,nat) -> nat

mult(n,m) =
 match n with | 0 -> 0
  | n+1 -> add(m,mult(n,m));
```

$$\Phi((n,m),(q_{(i,j)})) = \sum_{i,j\in\mathbb{N}} q_{(i,j)} \binom{n}{i}\binom{m}{j}$$

Number of evalua… he worst case: $8nm + 12n + 3$

Possible typing of `mult:` $(\mathrm{nat} * \mathrm{nat}, (q_{(i,j)})_{i,j\in\mathbb{N}}) \to (\mathrm{nat}, (p_i)_{i\in\mathbb{N}})$

where

$$\begin{aligned} q_{(0,0)} &= 3 \\ q_{(1,0)} &= 12 \\ q_{(1,1)} &= 18 \\ q_{(i,j)} &= 0 \text{ otherwise} \end{aligned}$$

$$\begin{aligned} p_1 &= 10 \\ p_i &= 0 \text{ otherwise} \end{aligned}$$

Example: Evaluation Steps of `mult`

```
add : (nat,nat) -> nat

add(n,m) =
 match n with | 0 -> m
  | n+1 -> 1+a
```

```
mult: (nat,nat) -> nat

mult(n,m) =
 match n with | 0 -> 0
  | n+1 -> add(m,mult(n,m));
```

$\Phi((n,m),(q_{(i,j)})) =$
$\sum_{i,j \in \mathbb{N}} q_{(i,j)} \binom{n}{i} \binom{m}{j}$

Number of evalua he worst case: $8nm + 12n + 3$

Possible typing of `mult`: $(\mathsf{nat} * \mathsf{nat}, (q_{(i,j)})_{i,j\in\mathbb{N}}) \to (\mathsf{nat}, (p_i)_{i\in\mathbb{N}})$

where

$$
\begin{aligned}
q_{(0,0)} &= 3 \\
q_{(1,0)} &= 12 \\
q_{(1,1)} &= 18 \\
q_{(i,j)} &= 0 \text{ otherwise}
\end{aligned}
\qquad
\begin{aligned}
p_1 &= 10 \\
p_i &= 0 \text{ otherwise}
\end{aligned}
$$

Output potential is consumed later.

Example: Evaluation Steps of `mult`

```
add : (nat,nat) -> nat

add(n,m) =
 match n with | 0 -> m
  | n+1 -> 1+a
```

```
mult: (nat,nat) -> nat

mult(n,m) =
 match n with | 0 -> 0
  | n+1 -> add(m,mult(n,m));
```

$$\Phi((n,m),(q_{(i,j)})) = \sum_{i,j\in\mathbb{N}} q_{(i,j)} \binom{n}{i}\binom{m}{j}$$

Number of evalua … he worst case: $8nm + 12n + 3$

Possible typing of `mult`: $(\text{nat} * \text{nat}, (q_{(i,j)})_{i,j\in\mathbb{N}}) \to (\text{nat}, (p_i)_{i\in\mathbb{N}})$

where

$$\begin{aligned} q_{(0,0)} &= 3 \\ q_{(1,0)} &= 12 \\ q_{(1,1)} &= 18 \\ q_{(i,j)} &= 0 \text{ otherwise} \end{aligned}$$

10nm = 10(mult(n,m))

$$\begin{aligned} p_1 &= 10 \\ p_i &= 0 \text{ otherwise} \end{aligned}$$

Output potential is consumed later.

How to Deal with Multiplications `x*y` ?

Code transformation to recursive function?

- Need to prove soundness (semantic and resource usage equivalence)
- Inefficient: a large constraint set is generated for each multiplication

Better approach: directly describe how to pass potential to the result

$$\Phi((n, m), Q) \geq \Phi(n \cdot m, Q') + \mathsf{cost}(mult)$$

How to Deal with Multiplications `x*y` ?

Code transformation to recursive function?

- Need to prove soundness (semantic and resource usage equivalence)
- Inefficient: a large constraint set is generated for each multiplication

Better approach: directly describe how to pass potential to the result

$$\Phi((n, m), Q) \geq \Phi(n \cdot m, Q') + \mathsf{cost}(mult)$$

Can we express this inequality with a succinct constraint system?

New Type Rule for Multiplication

$$\frac{Q = \boxdot(Q') + M^{\mathsf{mult}}}{x_1\mathord{:}\mathrm{nat}, x_2\mathord{:}\mathrm{nat}; Q \mathrel{\vert\!\!\!\overset{M}{\underline{}}} x_1 {*} x_2 : (\mathrm{nat}, Q')} \; (\mathrm{T{:}M\textsc{ult}})$$

$$\boxdot(Q) = (q'_{(i,j)})_{(i,j)\in\mathcal{I}(\mathrm{nat}*\mathrm{nat})} \qquad \text{if} \qquad q'_{(i,j)} = \sum_k A(i,j,k)\, q_k$$

New Type Rule for Multiplication

$$\frac{Q = \boxdot(Q') + M^{\mathsf{mult}}}{x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}; Q \stackrel{M}{\vdash} x_1 {*} x_2 : (\mathrm{nat}, Q')} \text{(T:Mult)}$$

$$\boxdot(Q) = (q'_{(i,j)})_{(i,j)\in\mathcal{I}(\mathrm{nat}*\mathrm{nat})} \qquad \text{if} \qquad q'_{(i,j)} = \sum_k A(i,j,k)\, q_k$$

$$\binom{nm}{k} = \sum_{i,j} A(i,j,k) \binom{n}{i}\binom{m}{j}$$

$$A(i,j,k) = \sum_{r,s} (-1)^{i+j+r+s} \binom{i}{r}\binom{j}{s}\binom{rs}{k} = \sum_n \frac{i!j!}{k!} S(n,i)\, S(n,j)\, s(k,n)$$

New Type Rule for Multiplication

$$\frac{Q = \boxdot(Q') + M^{\mathsf{mult}}}{x_1{:}\mathrm{nat}, x_2{:}\mathrm{nat}; Q \overset{M}{\vdash} x_1{*}x_2 : (\mathrm{nat}, Q')} \;(\mathrm{T{:}M{\small ULT}})$$

$$\boxdot(Q) = (q'_{(i,j)})_{(i,j)\in\mathcal{I}(\mathrm{nat}*\mathrm{nat})} \qquad \text{if} \qquad q'_{(i,j)} = \sum_k A(i,j,k)\, q_k$$

Riordan and Stein (1972)

$$\binom{nm}{k} = \sum_{i,j} A(i,j,k) \binom{n}{i}\binom{m}{j}$$

$$A(i,j,k) = \sum_{r,s} (-1)^{i+j+r+s} \binom{i}{r}\binom{j}{s}\binom{rs}{k} = \sum_n \frac{i!j!}{k!} S(n,i)\, S(n,j)\, s(k,n)$$

New Type Rule for Multiplication

Constant cost of multiplication.

$$\frac{Q = \boxdot(Q') + M^{\mathsf{mult}}}{x_1\text{:nat}, x_2\text{:nat}; Q \stackrel{M}{\vdash} x_1 * x_2 : (\text{nat}, Q')} \; (\text{T:Mult})$$

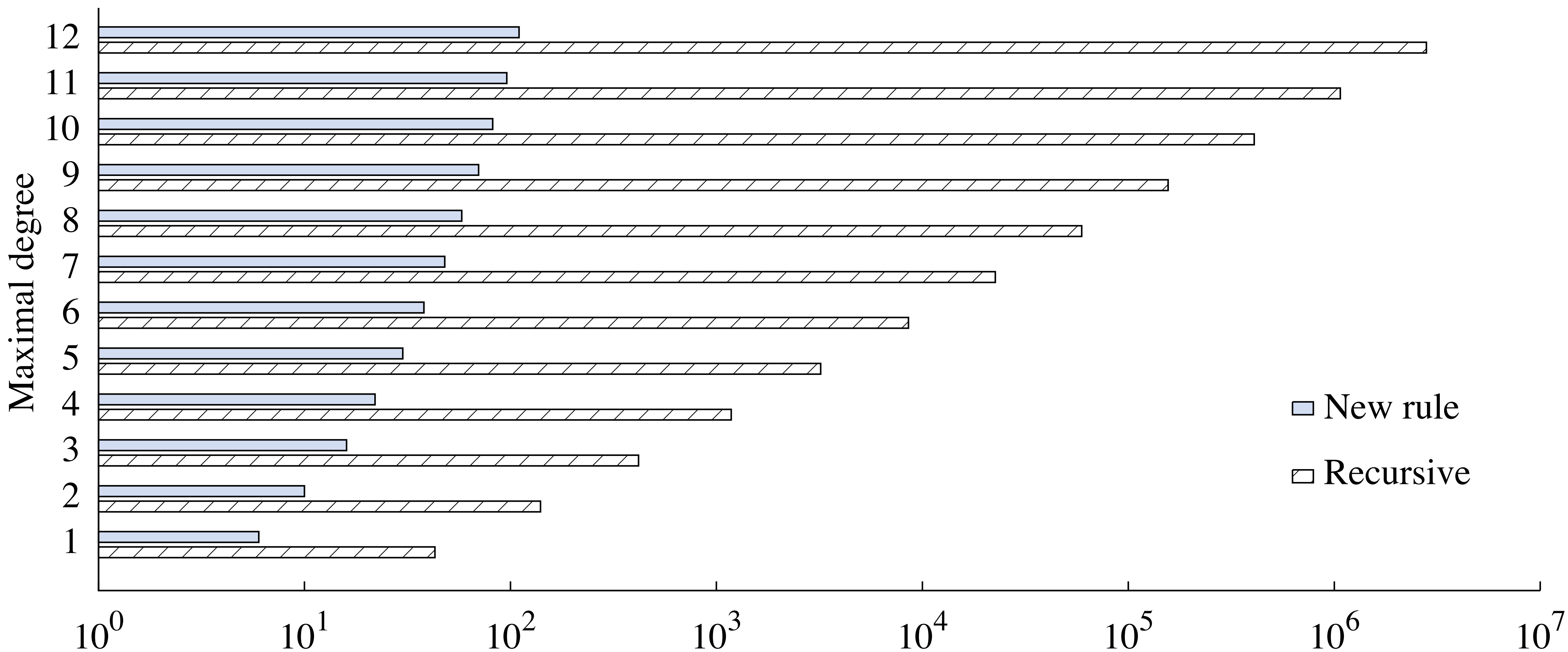
$$\boxdot(Q) = (q'_{(i,j)})_{(i,j)\in\mathcal{I}(\text{nat}*\text{nat})} \qquad \text{if} \qquad q'_{(i,j)} = \sum_k A(i,j,k)\, q_k$$

Riordan and Stein (1972)

$$\binom{nm}{k} = \sum_{i,j} A(i,j,k)\binom{n}{i}\binom{m}{j}$$

$$A(i,j,k) = \sum_{r,s}(-1)^{i+j+r+s}\binom{i}{r}\binom{j}{s}\binom{rs}{k} = \sum_n \frac{i!j!}{k!} S(n,i)\, S(n,j)\, s(k,n)$$

Smaller Constraints Sets Enable Scaling



Number of constraints generated for one multiplication

Other Arithmetic Operations

Treatment of other arithmetic operations is described in the paper

- Operations handled: subst, add, div, mod, mult
- Similar to multiplication

Also in the paper: arrays

- Arrays are treated as non-negative numbers: Array.length() returns a natural number that can be used for iteration
- Potential of data that is stored inside arrays is not tracked

How does it scale?

Dyadic Product of two Arrays

```
dyad : (Arr(int),nat,Arr(int),nat) -> Arr(Arr(int))

dyad (a,n,b,m) =
  let outerArr = A.make(n,A.make(0,+0)) in
  let _ = fill(a,n,b,m,outerArr) in outerArr;
```

Dyadic Product of two Arrays

```
dyad : (Arr(int),nat,Arr(int),nat) -> Arr(Arr(int))

dyad (a,n,b,m) =
  let outerArr = A.make(n,A.make(0,+0)) in
  let _ = fill(a,n,b,m,outerArr) in outerArr;
```

Computed evaluation-step bound:

$$20nm + 31n + 18$$

where

n is the value of the second component of the input

m is the value of the 4'th component of the input

Dyadic Product with Polynomials

```
matrix : (nat,nat) -> Arr(Arr(int))

matrix (n,m) =
  let size1 = n*n + 9*n + 28 in
  let size2 = m*n + 6*m in
  dyad( A.make(size1,+1),size1, A.make(size2,+1),size2 );
```

Dyadic Product with Polynomials

```
matrix : (nat,nat) -> Arr(Arr(int))

matrix (n,m) =
  let size1 = n*n + 9*n + 28 in
  let size2 = m*n + 6*m in
  dyad( A.make(size1,+1),size1, A.make(size2,+1),size2 );
```

Computed evaluation-step bound:

$$20mn^3 + 300mn^2 + 1641mn + 3366m + 32n^2 + 288n + 942$$

where
  n is the value of the first component of the input
  m is the value of the second component of the input

Dyadic Product with Polynomials

```
matrix : (nat,nat) -> Arr(Arr(int))

matrix (n,m) =
  let size1 = n*n + 9*n + 28 in
  let size2 = m*n + 6*m in
  dyad( A.make(size1,+1),size1, A.make(size2,+1),size2 );
```

Computes a $(n^2+9n+28) \times (mn+6m)$ matrix.

Computed evaluation-step bound:

$$20mn^3 + 300mn^2 + 1641mn + 3366m + 32n^2 + 288n + 942$$

where
n is the value of the first component of the input
m is the value of the second component of the input

Many Dyadic Products with Polynomials

```
dyadAllM : nat -> unit

dyadAllM n = match n with | 0 -> ()
               | S n' -> let _ = dyadM(n,n) in
                         dyadAllM(n');

dyadM(n,m) =  match m with | 0 -> ()
                | S m' -> let mat = matrix(n,m) in
                          dyadM(n,m');
```

Many Dyadic Products with Polynomials

```
dyadAllM : nat -> unit

dyadAllM n = match n with | 0 -> ()
                | S n' -> let _ = dyadM(n,n) in
                          dyadAllM(n');

dyadM(n,m) =  match m with | 0 -> ()
                 | S m' -> let mat = matrix(n,m) in
                           dyadM(n,m');
```

Computed evaluation-step bound:

$$1.66n^6 + 37n^5 + 334.79n^4 + 1485.08n^3 + 2963.54n^2 + 1789.92n + 3$$

where

n is the value of the input

Many Dyadic Products with Polynomials

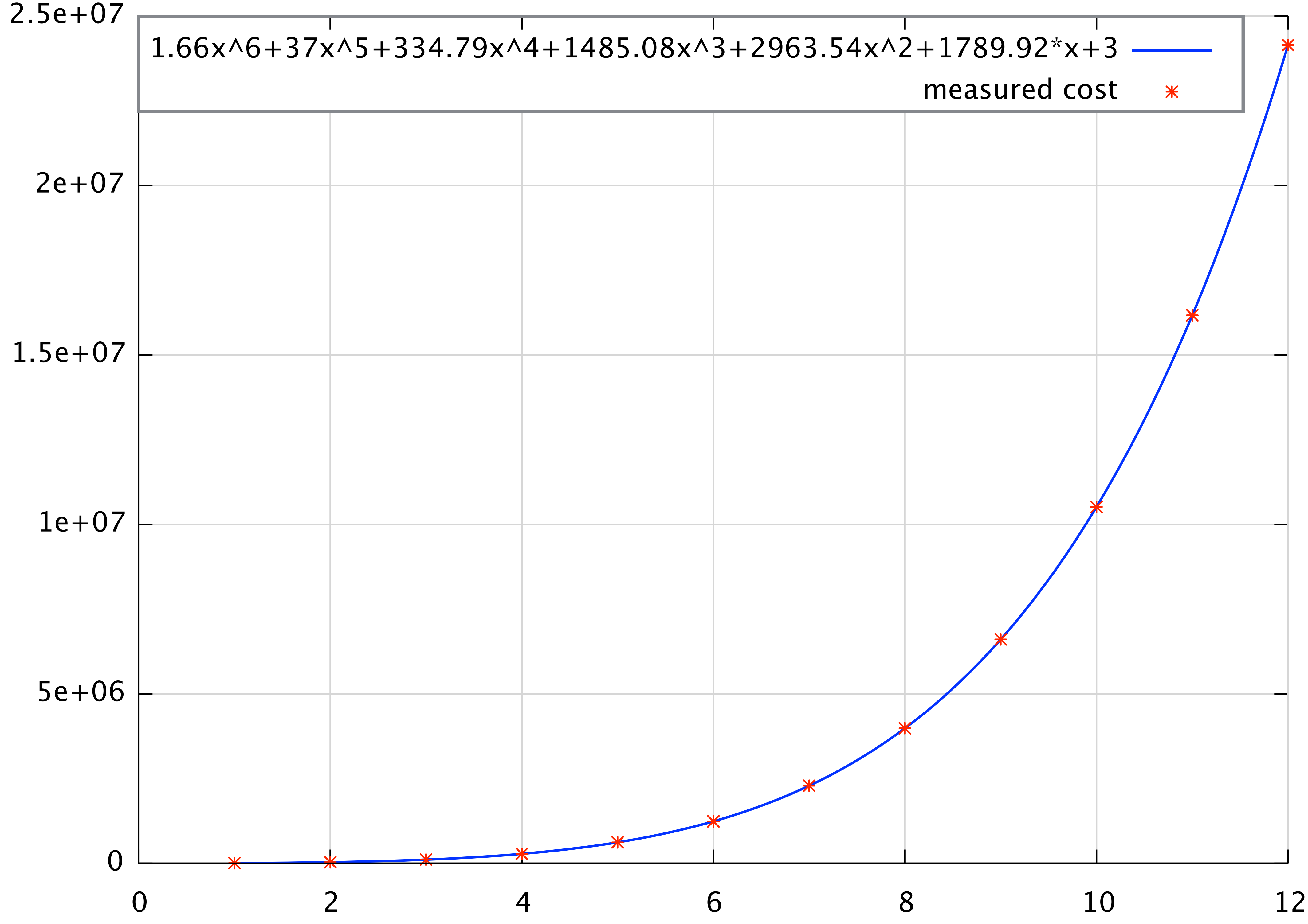
Computes a
$(i^2+9i+28) \times (ij+6j)$ matrix for every
pair (i,j) such that $1 \le j \le i \le n$.

```
dyadAllM : nat -> unit

dyadAllM n = match n with | 0 -> ()
               | S n' -> let _ = dyadM(n,n) in
                         dyadAllM(n');

dyadM(n,m) =  match m with | 0 -> ()
                | S m' -> let mat = matrix(n,m) in
                          dyadM(n,m');
```

Computed evaluation-step bound:

$$1.66n^6 + 37n^5 + 334.79n^4 + 1485.08n^3 + 2963.54n^2 + 1789.92n + 3$$

where
  n is the value of the input



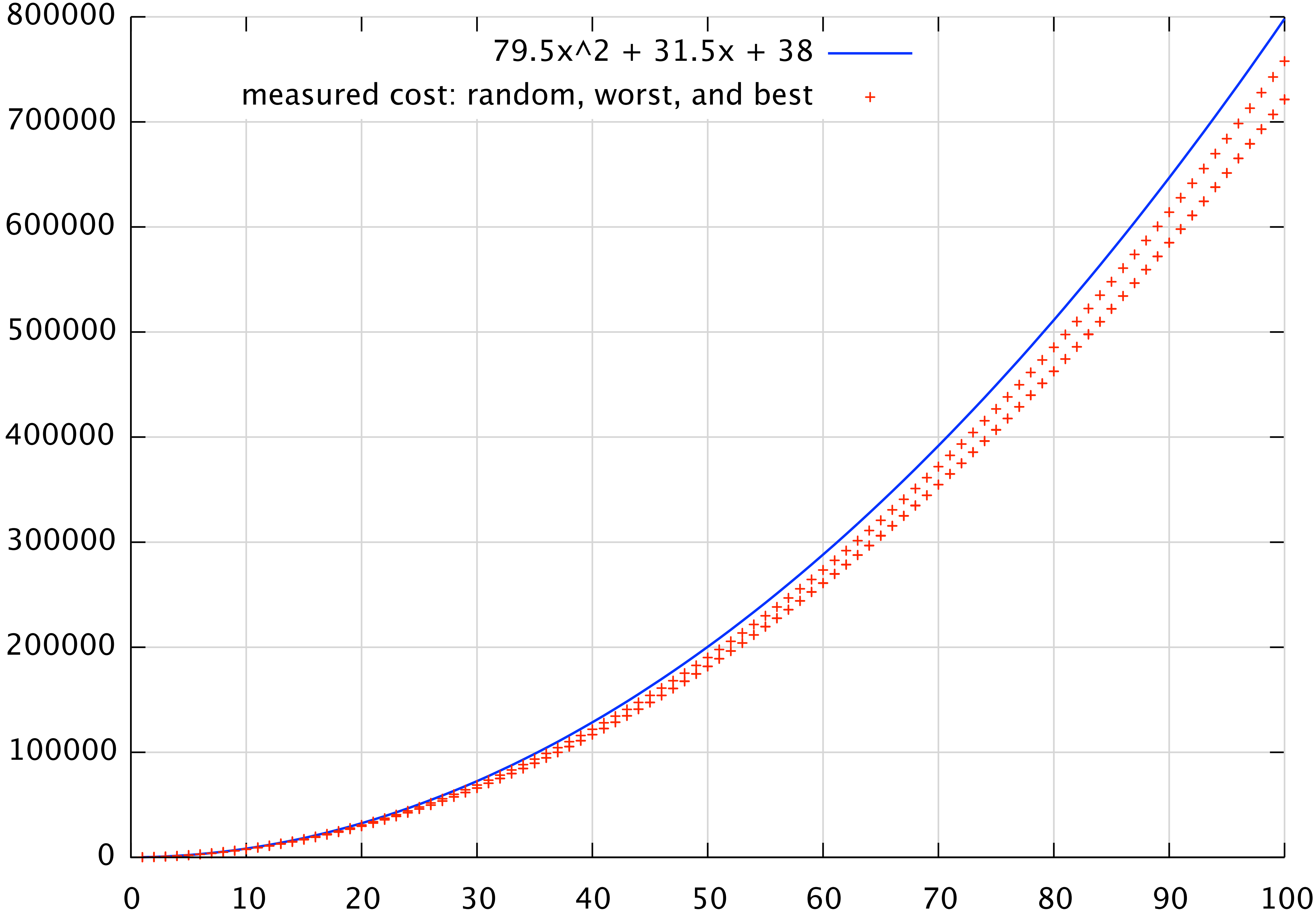
dyadAllM

Evaluation-step bound vs. measured behavior

Experimental Evaluation

	Computed Bound	Actual Behavior	Run Time	#Constr.
Dijkstra's Shortest Path	79.5n	O(n	0.1 s	2178
Fast GCD	12m + 7	O(log m)	0.1 s	105
Pascal's Triangle	19n	O(n	0.4 s	998
In-Place Quick Sort	12.25x	O(x	0.7 s	2080
Matrix Multiplication (for a list of matrices)	18nuyx + 31nuy + 38nu + 38n + 3	O(nuyx)	5.6 s	184270
Block Sort	12.25n	O(n	0.4 s	27795
DyadAllM	1.6n 2963.54n	O(n	3.9 s	130236
Matrix-Mult, Flatten, and Sort	12.25u + 19m + 66	O(u	5.9 s	167603

Evaluation-Step Bounds



Dijkstra’s Single-Source Shortest Path

Evaluation-step bound vs. measured behavior

Conclusion

Directly encoding (non-linear) arithmetic operations in amortized resource analysis lets us track size changes of unsigned integers precisely and efficiently.

Conclusion

Directly encoding (non-linear) arithmetic operations in amortized resource analysis lets us track size changes of unsigned integers precisely and efficiently.

Ongoing Research: Application of the amortized analysis to C programs

- Bounds are non-negative linear combin. of sizes of intervals |[x,y]|
- Great preliminary results for linear bounds
- Beats already abstract interpretation-based techniques
- Extension to polynomial bounds using the presented techniques