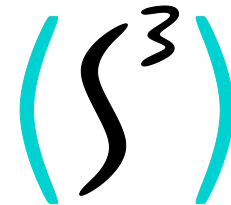


A Typed Interrupt Calculus



Jens Palsberg



Purdue University
Department of Computer Science

Joint work with Ma Di

Supported by an NSF ITR award and by DARPA.

(S³) **Secure Software Systems Group**

Faculty: Antony Hosking, Jens Palsberg, Jan Vitek

17 Ph.D. students

Current Support:

NSF

- 2 CAREER awards
- 2 ITR awards
- regular awards

DARPA

CERIAS

IBM

Intel

Lockheed Martin

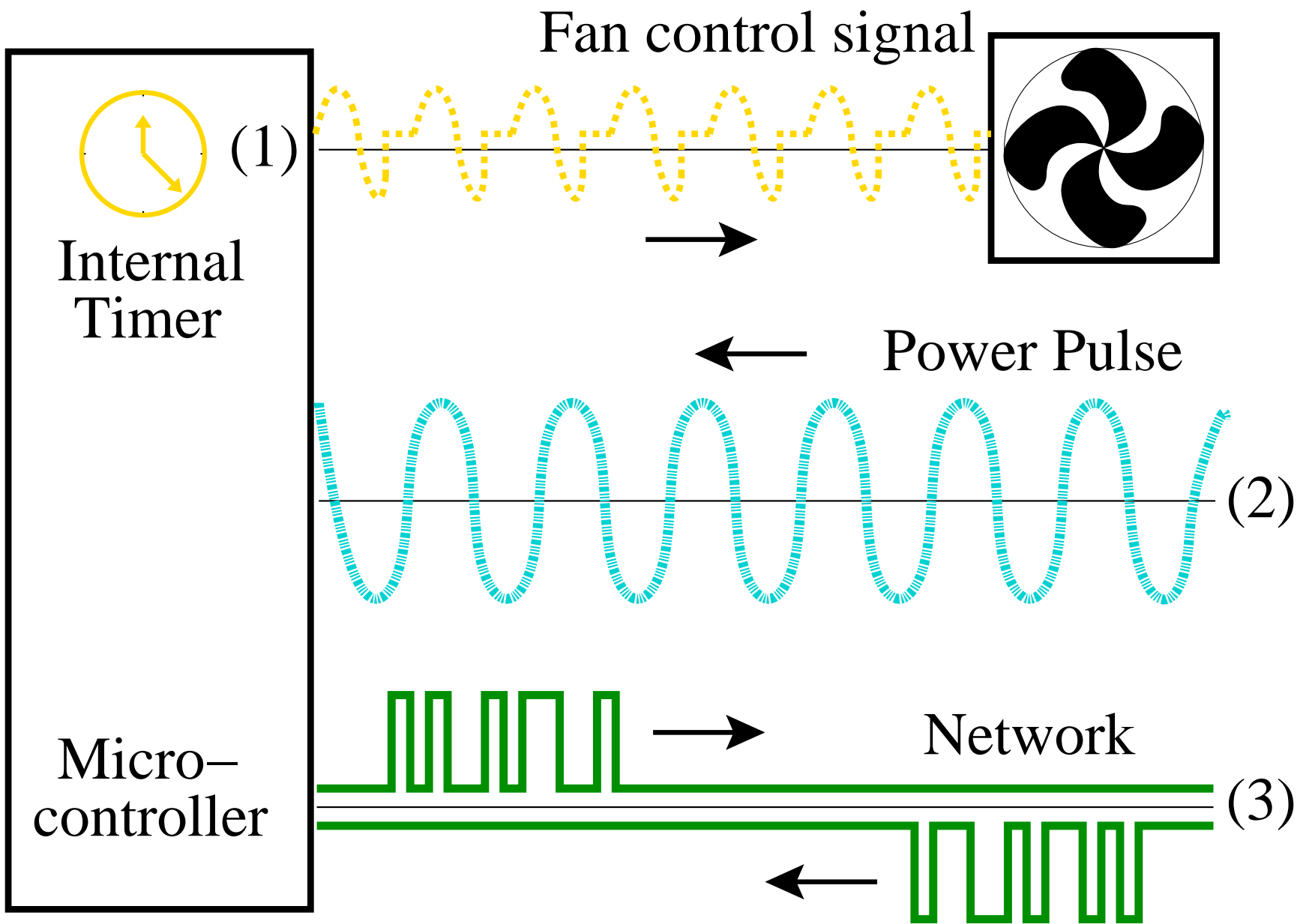
Microsoft

Motorola

Sun Microsystems

(§³) **Our Results**

- An interrupt calculus.
 - No program can terminate.
- A type system.
 - A well-typed program cannot cause stack overflow.
- A prototype implementation.



(S³) Example Program in Z86 Assembly Language

```
; Constant Pool (Symbol Table); Bit Flags for IMR and IRQ.
IRQ0 .EQU #00000001b
; Bit Flags for external devices on Port 0 and Port 3.
DEV2 .EQU #00010000b

; Interrupt Vectors.
    .ORG %00h
    .WORD #HANDLER ; Device 0

; Main Program Code.
    .ORG 0Ch
INIT: ; Initialization section.
0C LD SPL, #0F0h ; Initialize Stack Pointer.
0F LD RP, #10h ; Work in register bank 1.
12 LD P2M, #00h ; Set Port 2 lines to all outputs.
15 LD IRQ, #00h ; Clear IRQ.
18 LD IMR, #IRQ0
1B EI ; Enable Interrupt 0.
```

(S³) Example Program in Z86 Assembly Language

```
START:                ; Start of main program loop.
1C  DJNZ r2,  START ; If our counter expires,
1E  LD  r1,  P3    ; send this sensor's reading
20  CALL SEND     ; to the output device.
23  JP  START

SEND:                 ; Send Data to Device 2.
26  PUSH IMR      ; Remember what IMR was.
DELAY:
28  DI           ; Musn't be interrupted during pulse.
29  LD  P0,  #DEV2 ; Select control line for Device 2.
2C  DJNZ r3,  DELAY ; Short delay.
2E  CLR  P0
30  POP  IMR      ; Reactivate interrupts.
32  RET

HANDLER:             ; Interrupt for Device 0.
33  LD  r2,  #00h ; Reset counter in main loop.
35  CALL SEND
38  IRET          ; Interrupt Handler is done.
.END
```

(S³) Resource-Aware Compilation

A machine readable specification and an implementation:

Resource Constraints:

- Available code space: 512 KB
- Maximum stack size: 800 bytes
- Maximum time to handle event 1: 400 μ s
- Minimum battery life time: 2 years

Source Code:

// in a high-level language such as C

Can be compiled by a resource-aware compiler.

The generated assembly code can be verified by a model checker.

(§³) A Nasty Programming Error

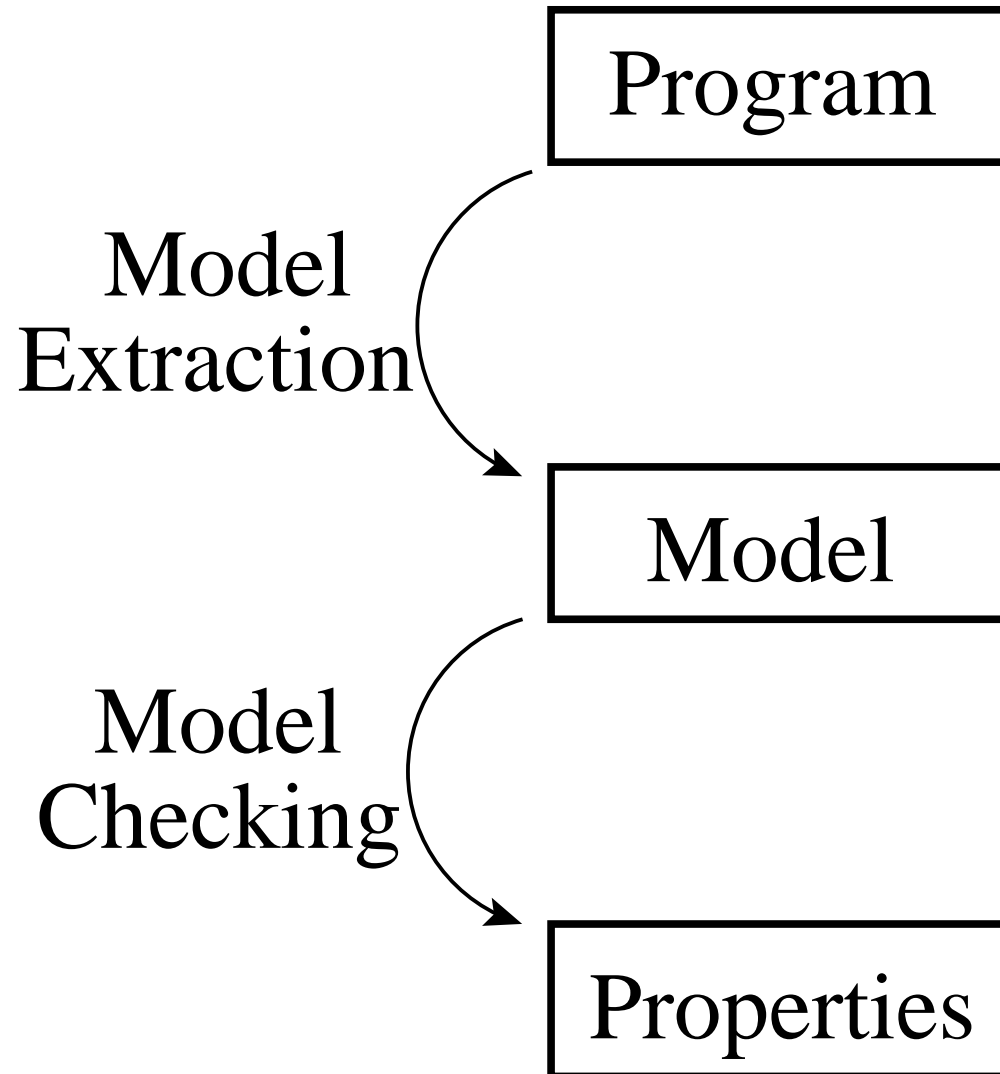
```
handler 1 {  
    // do something  
    enable-handling-of-interrupt-2  
    // do something else  
    iret  
}  
  
handler 2 {  
    // do something  
    enable-handling-of-interrupt-1  
    // do something else  
    iret  
}
```

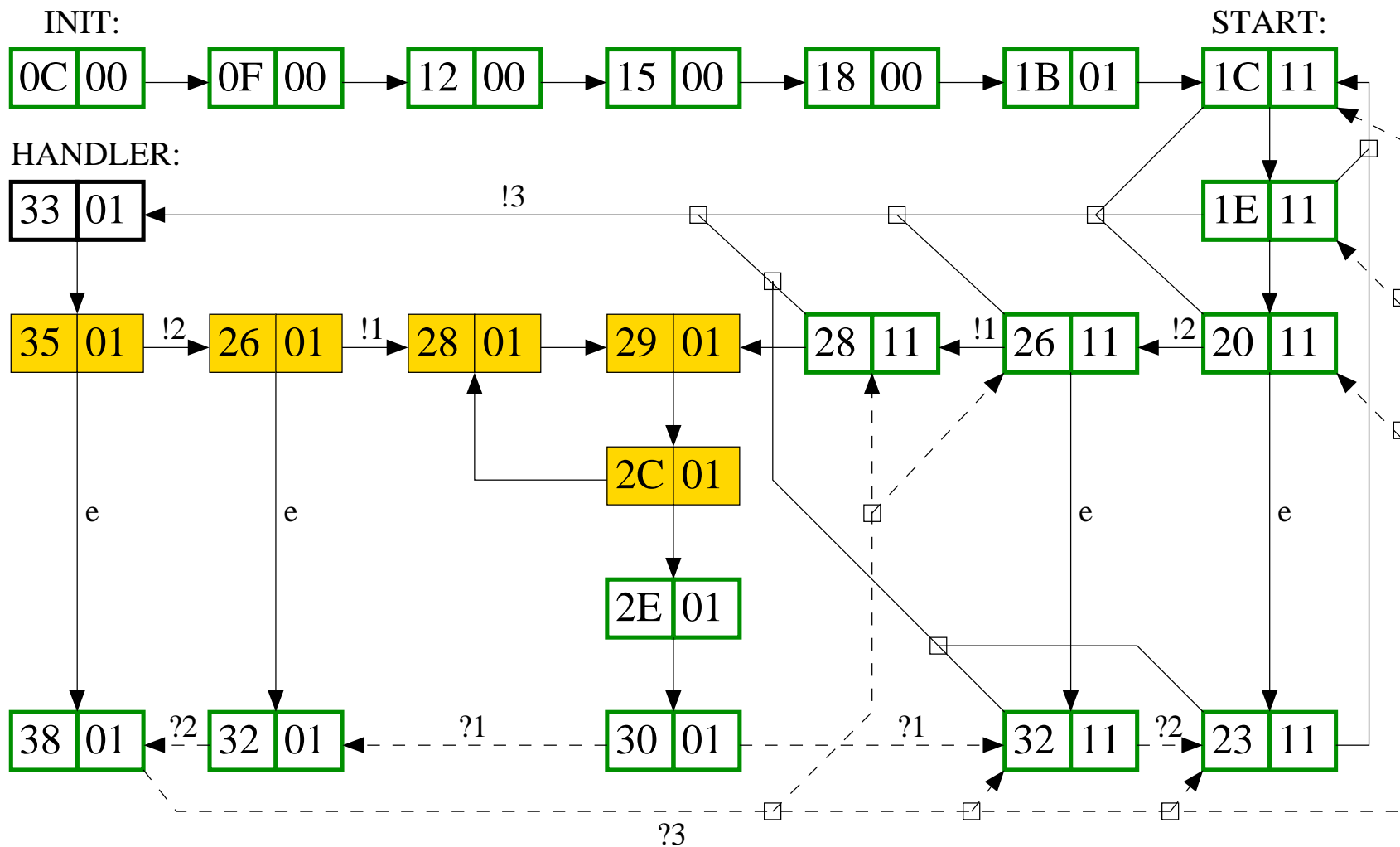
(§³) Interrupt Mask Register

| Well-known product | Processor | interrupt sources | master bit |
|--------------------|----------------------------------|-------------------|------------|
| Microcontroller | Zilog Z86 | 6 | yes |
| iPAQ Pocket PC | Intel strongARM, XScale | 21 | no |
| Palm | Motorola Dragonball (68K Family) | 22 | yes |
| Microcontroller | Intel MCS-51 Family (8051 etc) | 6 | yes |

MCS-51 interrupt mask register:

| | | | | | | | |
|----|---|-----|----|-----|-----|-----|-----|
| EA | – | ET2 | ES | ET1 | EX1 | ET0 | EX0 |
|----|---|-----|----|-----|-----|-----|-----|





(S³) Stack-Size Analysis

| Program | Lower | Upper |
|---------|-------|-------|
| CTurk | 17 | 18 |
| GTurk | 16 | 17 |
| ZTurk | 16 | 17 |
| DRop | 12 | 14 |
| Rop | 12 | 14 |
| Fan | 11 | N/A |
| Serial | 10 | 10 |

The lower bounds were found with a software simulator for Z86 assembly language that we wrote.

(§³) Two Selfish Handlers

```
maximum stack size: 1
```

```
imr = imr or 111b
```

```
loop {
```

```
    skip
```

```
    imr = imr or 111b
```

```
}
```

```
handler 1 [ ( 111b -> 111b : 0 )  
           ] {
```

```
    skip
```

```
    iret
```

```
}
```

```
handler 2 [ ( 111b -> 111b : 0 )  
           ] {
```

```
    skip
```

```
    iret
```

```
}
```

(§³) Two Prioritized Handlers

```

maximum stack size: 2

imr = imr or 111b

loop {
    skip
    imr = imr or 111b
}

handler 1 [ ( 111b -> 111b : 0 )
           ( 110b -> 110b : 0 )
         ] {
    skip
    ired
}

handler 2 [ ( 111b -> 111b : 1 )
         ] {
    skip
    imr = imr and 110b
    imr = imr or 100b
    ired
}
```

(S³) Two Cooperative Handlers

maximum stack size: 2

```
imr = imr or 111b
loop {
    imr = imr or 111b
}
```

```
handler 1 [ ( 111b -> 101b : 1 )
           ( 110b -> 100b : 0 )
           ] {
    imr = imr and 101b
    imr = imr or 100b
    iret
}
handler 2 [ ( 111b -> 110b : 1 )
           ( 101b -> 100b : 0 )
           ] {
    imr = imr and 110b
    imr = imr or 100b
    iret
}
```

(S³) Two Fancy Handlers

maximum stack size: 3

```
imr = imr or 111b
loop {
    imr = imr or 111b
}
```

```
handler 1 [ ( 111b -> 111b : 2 )
            ( 110b -> 100b : 0 )
          ] {
    imr = imr and 101b
    imr = imr or 100b
    ired
}
handler 2 [ ( 111b -> 100b : 1 )
            ( 101b -> 100b : 1 )
          ] {
    imr = imr and 110b
    imr = imr or 010b
    imr = imr or 100b
    imr = imr and 101b
    ired
}
```

(S³) A Timer

```
maximum stack size: 1

SEC = SEC + 60
imr = imr or 110b
loop {
    if( SEC == 0 ) {
        OUT = 1
        imr = imr and 101b
        imr = imr or 001b
    } else {
        OUT = 0
    }
}

handler 1 [ ( 111b -> 111b : 0 )
           ( 110b -> 110b : 0 )
         ] {
    SEC = SEC + (-1)
    ired
}

handler 2 [ ( 111b -> 110b : 0 )
           ( 101b -> 110b : 0 )
         ] {
    SEC = 60
    imr = imr and 110b
    imr = imr or 010b
    ired
}
```

(S³) The Interrupt Calculus

(program) $p ::= (m, \bar{h})$

(main) $m ::= \text{loop } s \mid s ; m$

(handler) $h ::= \text{iret} \mid s ; h$

(statements) $s ::= x = e \mid \text{imr} = \text{imr} \wedge \text{imr} \mid \text{imr} = \text{imr} \vee \text{imr} \mid$
 $\text{if0 } x \text{ then } s_1 \text{ else } s_2 \mid s_1 ; s_2 \mid \text{skip}$

(expression) $e ::= c \mid x \mid x + c \mid x_1 + x_2$

(§³) Operational Semantics

Handlers \bar{h} , store R , interrupt mask register imr , stack σ , action a .

$$\langle \bar{h}, R, imr, \sigma, a \rangle \rightarrow \langle \bar{h}, R, imr \wedge \mathbf{t}_0^\bullet, a :: \sigma, \bar{h}(i) \rangle$$

if *enabled(imr, i)*

$$\langle \bar{h}, R, imr, \sigma, iret \rangle \rightarrow \langle \bar{h}, R, imr \vee \mathbf{t}_0, \sigma', a \rangle \text{ if } \sigma = a :: \sigma'$$

$$\langle \bar{h}, R, imr, \sigma, imr = imr \wedge imr'; a \rangle \rightarrow \langle \bar{h}, R, imr \wedge imr', \sigma, a \rangle$$

$$\langle \bar{h}, R, imr, \sigma, skip; a \rangle \rightarrow \langle \bar{h}, R, imr, \sigma, a \rangle$$

Theorem: No program can terminate

(S³) Type Judgments

$$\tau \equiv \bigwedge_{j=1}^n ((\widehat{imr})^j \xrightarrow{\delta^j} (\widehat{imr}')^j).$$

| Type Judgment | Meaning |
|---|---------------------------------------|
| $\bar{\tau} \vdash h : \tau$ | Interrupt handler h has type τ |
| $\bar{\tau}, \widehat{imr} \vdash_K \sigma$ | Stack σ type checks |
| $\bar{\tau}, \widehat{imr} \vdash_K m$ | Main part m type checks |
| $\bar{\tau}, \widehat{imr} \vdash_K h : \widehat{imr}'$ | Handler h type checks |
| $\bar{\tau}, \widehat{imr} \vdash_K s : \widehat{imr}'$ | Statement s type checks |
| $\bar{\tau} \vdash_K P$ | Program state P type checks |

(§³) Type Rules

$$\frac{\bar{\tau}, (\widehat{imr})^j \wedge \mathbf{t}_0^\bullet \vdash_{\delta^j} h : (\widehat{imr}')^j \quad j \in 1..n}{\bar{\tau} \vdash h : \bigwedge_{j=1}^n ((\widehat{imr})^j \xrightarrow{\delta^j} (\widehat{imr}')^j)}$$

$$\frac{\bar{\tau}, \widehat{imr} \vdash_K s_1 : \widehat{imr}_1 \quad \bar{\tau}, \widehat{imr}_1 \vdash_K s_2 : \widehat{imr}_2}{\bar{\tau}, \widehat{imr} \vdash_K s_1; s_2 : \widehat{imr}_2}$$

$$\bar{\tau}, \widehat{imr} \vdash_K \text{skip} : \widehat{imr} \quad \left[\text{safe}(\bar{\tau}, \widehat{imr}, K) \right]$$

$$\text{safe}(\bar{\tau}, \widehat{imr}, K) = \left[\begin{array}{l} \forall i \in 1 \dots n \\ \text{if } \text{enabled}(\widehat{imr}, i) \\ \text{then, whenever } \bar{\tau}(i) = \dots \wedge (\widehat{imr} \xrightarrow{\delta} \widehat{imr}') \wedge \dots, \\ \text{we have } \widehat{imr}' \leq \widehat{imr} \wedge \delta + 1 \leq K \end{array} \right].$$

Theorem: A well-typed program cannot cause stack overflow

(§³) Conclusion

Calculus + type system + stack boundedness + prototype.

Future work: type inference + experiments.

High-assurance embedded systems in high-level languages =

machine-readable specifications

- + type systems
- + model checking
- + time-, space-, and power-aware compiler
- + automatic testcase generation.

[Brylow, Damgaard & Palsberg, ICSE 2001]: model checking

[Naik & Palsberg, LCTES 2002]: space-aware compilation

[Palsberg & Ma, FTRTFT 2002]: stack boundedness

[Palsberg & Wallace, manuscript]: reverse engineering