# PICOBIT: A Compact Scheme System for Microcontrollers

Vincent St-Amour
Université de Montréal
(now at Northeastern University)

Marc Feeley
Université de Montréal

21st Symposium on the Implementation and Application of Functional Languages
September 23, 2009

# Outline

- Motivation: small embedded systems
- System components
  - The PICOBIT Scheme compiler
  - The PICOBIT virtual machine
  - The SIXPIC C compiler
- Experimental results
- Future work

# Small embedded systems


- High volume
- Low cost ($1-$5 per microcontroller)
- Low memory (8-32 kB ROM 1-4 kB RAM)
- Low computational power (10 MIPS, 10 mW)
- Think microwave ovens, simple robots

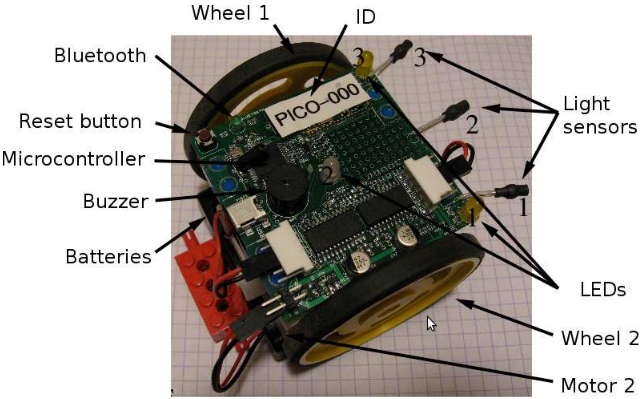
# Present state of affairs

- C or assembly
- Low level of abstraction
- Manual memory management
- Unsafe

# Enter PICOBIT

- Scheme
- Automatic memory management
- Closures and higher-order functions
- First-class continuations
- Lightweight threads
- Built-in data structures
- Bignums
- Safety

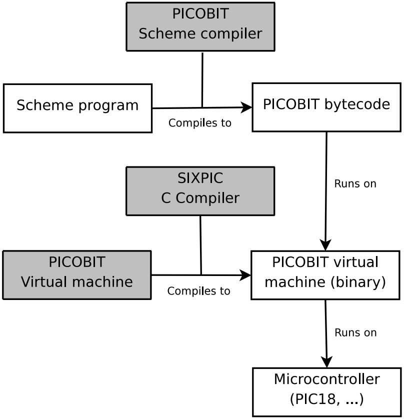
# The PICOBOARD robot



# Goals

- Complex applications
- Low speed requirements
- Low memory footprint
- Compact code

# Overview




- PICOBIT Scheme compiler
  - Written in Scheme
  - Compact custom instruction set
- PICOBIT virtual machine
  - Written in C
  - Highly portable
- SIXPIC C compiler
  - Written in Scheme
  - Designed for VMs

# General approach

- Omit useless features
- Optimizations for code size
- High-level bytecode
- Controlling the whole pipeline
  - Adapt the bytecode
  - Domain-specific optimizations

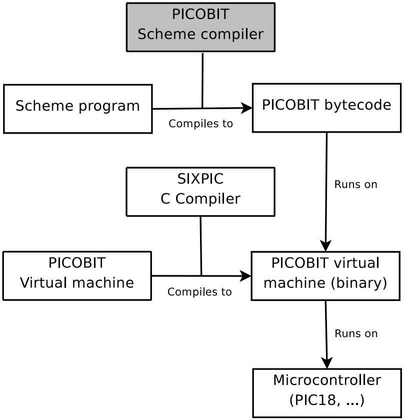
# Different flavors

- Full PICOBIT
  - 15.6 kB
- PICOBIT without bignums
  - 11.6 kB
- PICOBIT Light
  - 5.2 kB
  - No bignums
  - No byte vectors
  - Limited to 16 global variables
  - Limited to 128 heap objects

# The PICOBIT Scheme compiler

# The PICOBIT Scheme Compiler

- Scheme to bytecode
- Whole-program compilation
- Selected optimizations
- Custom instruction set



# Dead weight

- Floating-point numbers
- File I/O
- `eval`
- S-expression input

# Optimizations

- Mutability analysis
- Trace scheduling
- Treeshaker
  - Standard library : 2064 bytes
    - Strings : 508 bytes
    - Networking : 257 bytes
    - Threads : 141 bytes
  - Remote control : 106 bytes

# Custom instruction set

- Designed for compactness
- Short and long instruction encodings

| | |
|---|---|
| `000xxxxx` | Push constant $x$ |
| `1010xxxx xxxxxxxx` | Push constant $x$ |
| `1001xxxx` | Go to address $pc + x$ if TOS is false |
| `10111000 xxxxxxxx` | Go to address $pc + x - 128$ if TOS is false |
| `10110011 xxxxxxxx xxxxxxxx` | Go to address $x$ if TOS is false |

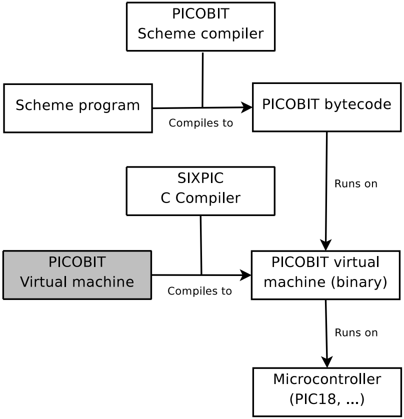
- Short encodings for frequent values
- Short encodings for frequent instructions
- High-level instructions

| | |
|---|---|
| `10111001 xxxxxxxx` | Build a closure with entry point $pc + x - 128$ |
| `11101100` | Copy data between the 2 byte vectors on TOS |
| `11110011` | Receive network packet to the byte vector on TOS |

# The PICOBIT virtual machine

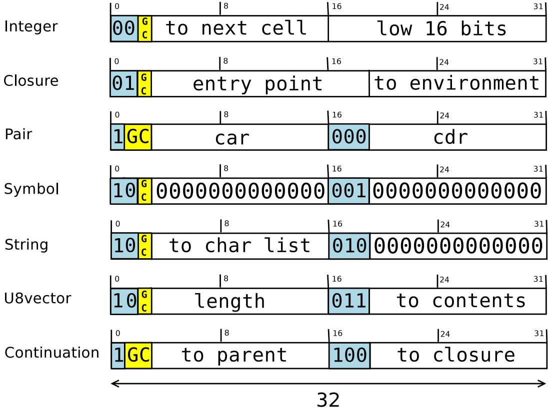
# The PICOBIT Virtual Machine

- Designed to be compact
- Simple data structures and algorithms



# Object encodings

- Data stack and continuations allocated in the heap
- Symbols are addresses



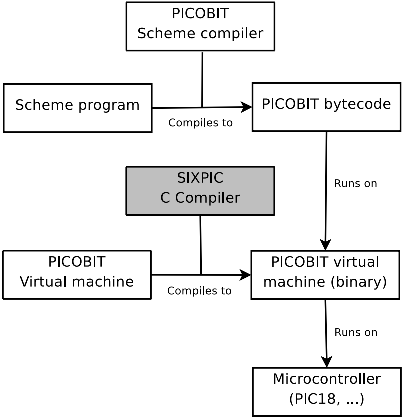
# Automatic Memory Management

- Mark-and-sweep
  - Simple algorithm, compact to implement
  - Compact because no to-space is needed
- Deutsche-Schorr-Waite's algorithm (pointer reversal)
  - No need to allocate a stack
  - More room for the heap

# The SIXPIC C compiler

# The SIXPIC C Compiler

- Compiles our VM
- Omits some features of C
- Selected optimizations



# Dead weight

- Floating-point numbers
- Signed numbers
- Structs and unions
- Function pointers
- Recursion

# Calling convention

- Arguments moved directly to the callee's local variables
- Whole-program register allocation
- 875 function calls in PICOBIT
- Saves 29.2%

```
byte f (byte x);
...
f(y);
```

---

```
   push $y      ; 8b
   call $f      ; 4b
...
f: pop  $x      ; 8b   total: 20 bytes
```

---

```
   move  $y A   ; 4b
   call  $f     ; 4b
...
f: move  A $x   ; 4b   total: 12 bytes
```

---

```
   move  $y $x ; 4b
   call  $f    ; 4b
...
f:             ;      total: 8  bytes
```

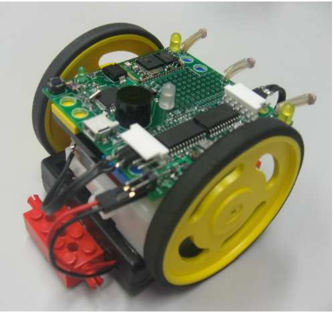
# Optimizations

- Register coalescing
  - Saves 4.1%
  - Plays well with our calling convention
  - 2420 byte cells
  - 1453 byte cells coalesced
  - 324 bytes of RAM
- Trace scheduling
  - Saves 6.3%
  - 519 jumps shortened
  - 228 jumps eliminated
- Treeshaker

# Experimental results

# Experimental Results

| | |
|---|---|
| Flashing led | 9 B |
| Follow the light | 101 B |
| Remote control | 106 B |
| Hello | 355 B |
| Light sensors | 374 B |
| Multi-threaded presence counter | 599 B |
| Web server | 1033 B |



| Network Stack | Stack size (kB) | VM size (kB) | Total size (kB) |
|---|---|---|---|
| $S^3$ | 3.1 | 15.6 | 18.7 |
| uIP | 10.0 | - | 10.0 |

# Experimental Results

| Version | MCC18 | SIXPIC | Hi-Tech C |
|---|---|---|---|
| Full PICOBIT | 24.8 kB | 17.5 kB | 15.6 kB |
| Without bignums | 17.0 kB | 13.0 kB | 11.6 kB |
| PICOBIT Light | 8.0 kB | 7.2 kB | 5.2 kB |

- Mostly the same restrictions for all 3
- SIXPIC outperforms MCC18 by about 42%
- Hi-Tech C outperforms SIXPIC by about 12%

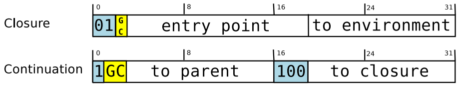
# Future work

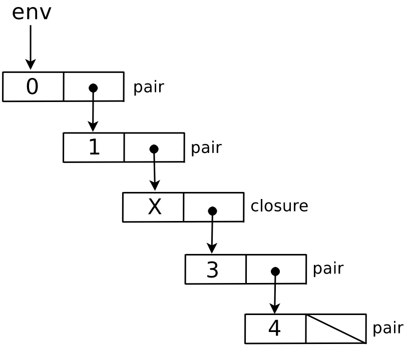
# Future Work

- Automatic procedural abstraction
- Huffman-encoded bytecode
- Exploit more virtual machine properties
- More general-purpose optimizations
- Port more languages

# Conclusion

- Compact Scheme system
- Can compete with C in terms of code size
- Can fit in less than 20 kB of ROM




```scheme
(define root-k #f) ;; root (empty) continuation
(define readyq #f) ;; queue of runnable threads

(define (start-first-process thunk)
  (set! root-k (get-cont))
  (set! readyq (cons #f #f))
  (set-cdr! readyq readyq)
  (thunk))

(define (spawn thunk)
  (let* ((k (get-cont))
         (next (cons k (cdr readyq))))
    (set-cdr! readyq next)
    (graft-to-cont root-k thunk)))

(define (exit)
  (let ((next (cdr readyq)))
    (if (eq? next readyq)
        (halt)
        (begin (set-cdr! readyq (cdr next))
               (return-to-cont (car next) #f)))))

(define (yield)
  (let ((k (get-cont)))
    (set-car! readyq k)
    (set! readyq (cdr readyq))
    (let ((next-k (car readyq)))
      (set-car! readyq #f)
      (return-to-cont next-k #f))))
```