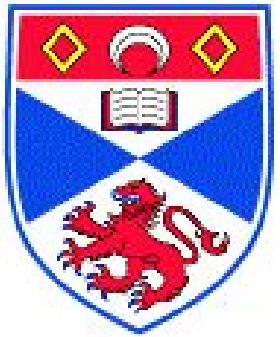


University of St Andrews

From Forth to Tay:
A component-based extensible virtual machine for compact programs

Simon Dobson
School of Computer Science, University of St Andrews UK

simon.dobson@st-andrews.ac.uk
http://www.simondobson.org

Overview

- Programming for WSNs
  - Nodes not governed by Moore's law
  - Radically distributed, radically failure-prone
- Languages need re-thinking
  - New control constructs, global behaviour

- This talk
  - A idea for compact, extensible VMs

A brief history of computing

- Back in the day, we lacked a certain something
  - Memory, mainly
  - But also compute power, storage, IDEs, high-resolution graphics, windowing systems, operating systems, memory management, optimising compilers, type checking, objects, debuggers, the internet, portable code, …
- Then we built big compilers, bigger computers, and hooked them together. All problems were solved, and everything was wonderful.
  The end.

Or was it?...

Abnormal environments

- Our ideas of software development have been conditioned by Moore's law
  - Larger, more complex environments generating flexible code
- What happens when this isn't the case?
  - Can't assume platforms will *ever* get substantially better

- What happens when we're coding in what isn't a “normal” environment?
  - Maintain sensible behaviour in the sense of distinctly non-sensible inputs and partial failure

Targeting a new domain

Presumably you have some set of tasks, some domain of application, for which you think a new programming language would be better than any existing language.

Think about what people want to be able to say. What are the problems, the applications, that this programming language is going to be used for? How would you like to express them in that language?

What would be the most natural way to write them down?

What are the most important examples, the simplest ones that would get somebody started? Try to make those as straightforward as possible.

Brian Kernighan, quoted in Biancuzzi and Warden. Masterminds of programming. O'Reilly Media. 2009.

The new domain

- Sensor networks are clearly different, but we're unclear as to what this means
  - New types: uncertainty, ranges, probability, ...
  - New control structures: decision-making by reasoning, rollback, …
  - New composition modes: failure is normal, resource discovery, …
- All suggest new languages – but what?
  - Because we don't know, we need to experiment
  - Extend/change features on offer

Extension – 1

- Layering
  - Syntactic and/or semantic sugaring
  - Raise the abstraction level, but not necessarily for the compiler

```
while(x>10) ...
```

```
test: if(!(x>10)
         goto loop;
      …
      goto test;
loop:
```

- Libraries
  - No new types or syntax

```
Semaphore s, t, u;
P(s);
   P(t);
   …
   if(...) V(t) else …
   …
   V(t);
V(s);
```

Functionally, monitors and `synchronized` blocks provide no advantage over semaphores. However…

Extension – 2

- Syntax
  - Re-write terms *without reducing*
  - Need language support, still looks like Scheme

```
(define-macro when
  (lambda (test . branch)
    (list 'if test
      (cons 'begin branch))))
```

These variables are *pattern* variables that aren't expanded until the macro is used

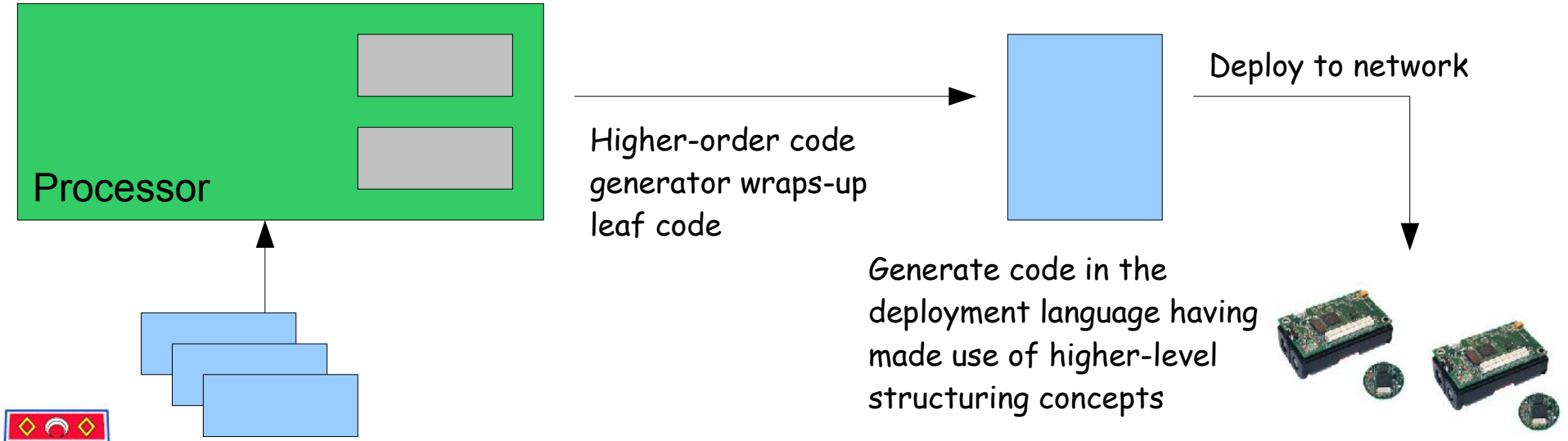
```
(when (< (pressure tube) 60)
   (open-valve tube)
   (attach floor-pump tube)
   (depress floor-pump 5)
   (detach floor-pump tube)
   (close-valve tube))
```

```
(apply
  (lambda (test . branch)
    (list 'if test
      (cons 'begin branch)))
  '((< (pressure tube) 60)
     (open-valve tube)
     (attach floor-pump tube)
     (depress floor-pump 5)
     (detach floor-pump tube)
     (close-valve tube)))
```

Staged compilation

- Use one language to structure another
- E.g. Flask: stage NesC *via* Haskell
  - Leaf functions get deployed
  - Structuring and data pathways done by higher-order functions that act as code generators

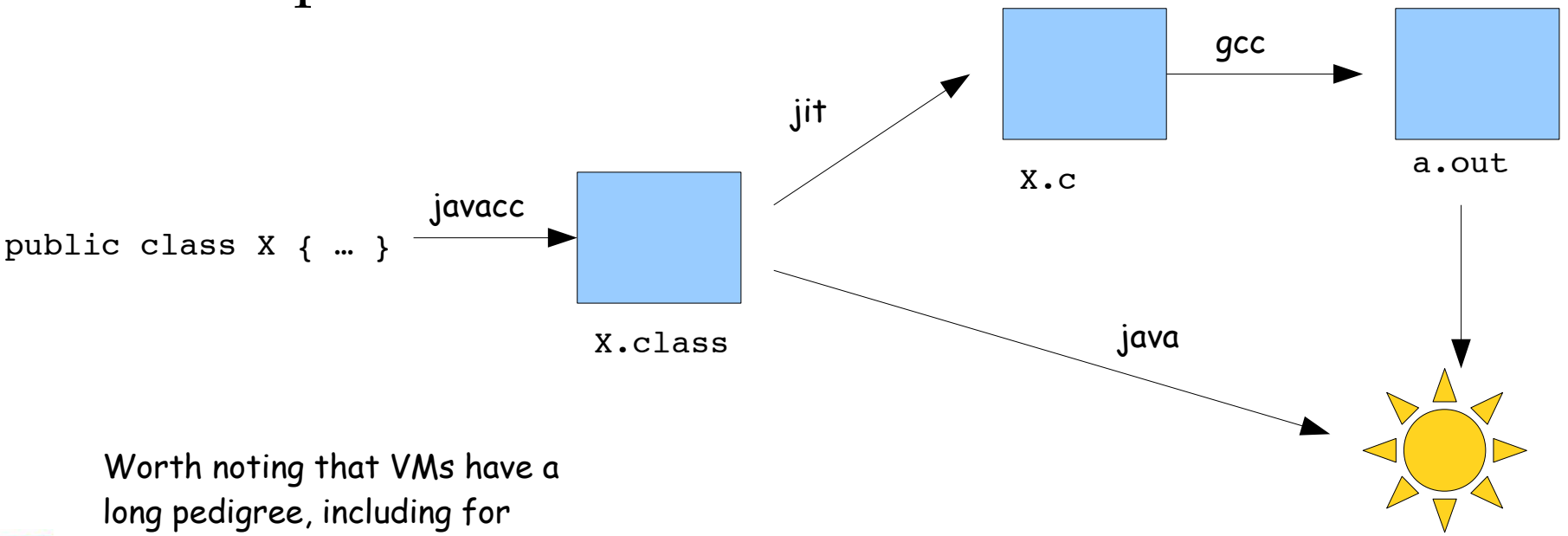
Mainland *et alia*. Flask: staged functional programming for sensor networks. Proc. ICFP. 2008



Virtual machines

- Canonical example is the Java VM
  - A machine whose machine code is in some sense optimised for Java
  - Provide a small, tight interpreter ported to each "real" platform

Also target other languages at it, for example Scala



Worth noting that VMs have a long pedigree, including for Smalltalk, Pascal, .NET

The VM inner loop

- Interpret the bytecodes
  - Decode instruction
  - Switch on the opcode
  - A small, fixed population

Bytecode may be "packed" with offsets or other literal information, as well as instruction opcode

```
void bytecode_inner_interpreter() {
    while(TRUE) {
      bytecode = *ip++;
      opcode = unpack_opcode(bytecode);
      switch(opcode) {
        case OP_NOOP:
          break;
        ...
        case OP_JUMP:
          ip += unpack_offset(bytecode);
          break
        ...
      }
    }
}
```

Designing the VM – 1

- For Java
  - Bytecodes for a stack machine
  - Arithmetic, object creation, method call, slot update, object lock/unlock
  - (Some peculiar omissions: no bytecodes create or control threads, for example)
- A single set of decisions, made once
  - Does this make sense in the modern world?

Designing the VM – 2

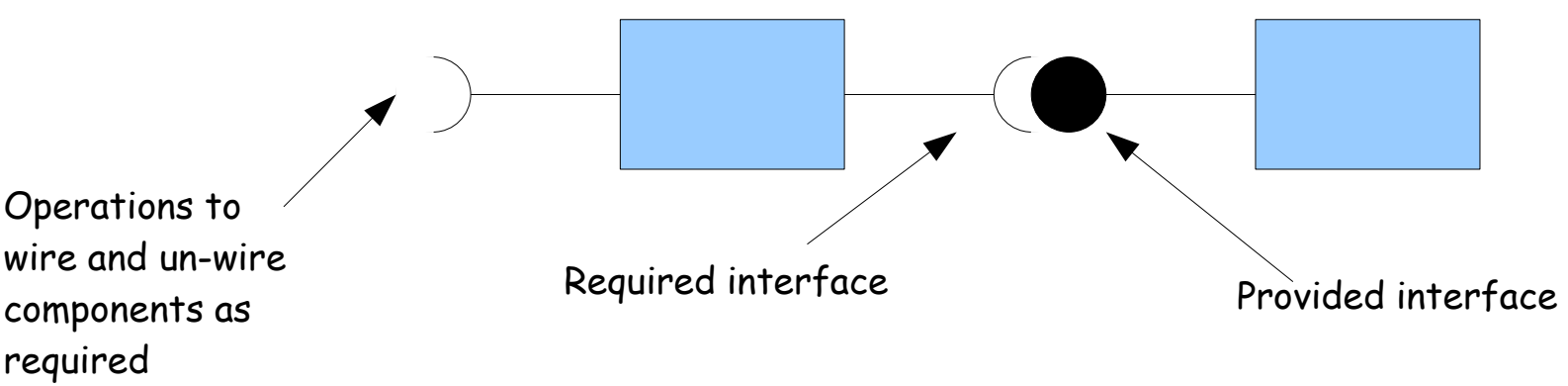
- Engineering decisions made early and then fossilised: now almost impossible to change
  - "Profiles" outlaw some programs, i.e. real numbers on small platforms
- Limited bytecode re-writing
  - Some JVMs for sensor networks provide "tighter" bytecode to reduce radio traffic
- Refactor some functions
  - Move verification to gateway, nodes can assume a "trusted compiler"

Extensible VMs

- What if we *didn't* fix the VM?
  - Allow for changes in the bytecode (or whatever)
  - Let the VM evolve alongside the rest of the codebase
- Add – and, crucially, remove – features
  - Removal is crucial: get rid of problematic constructions, don't just wish them away
  - Keep the core, extend in a structured way
  - Migrate what we know about good software engineering all the way down

Components all the way down

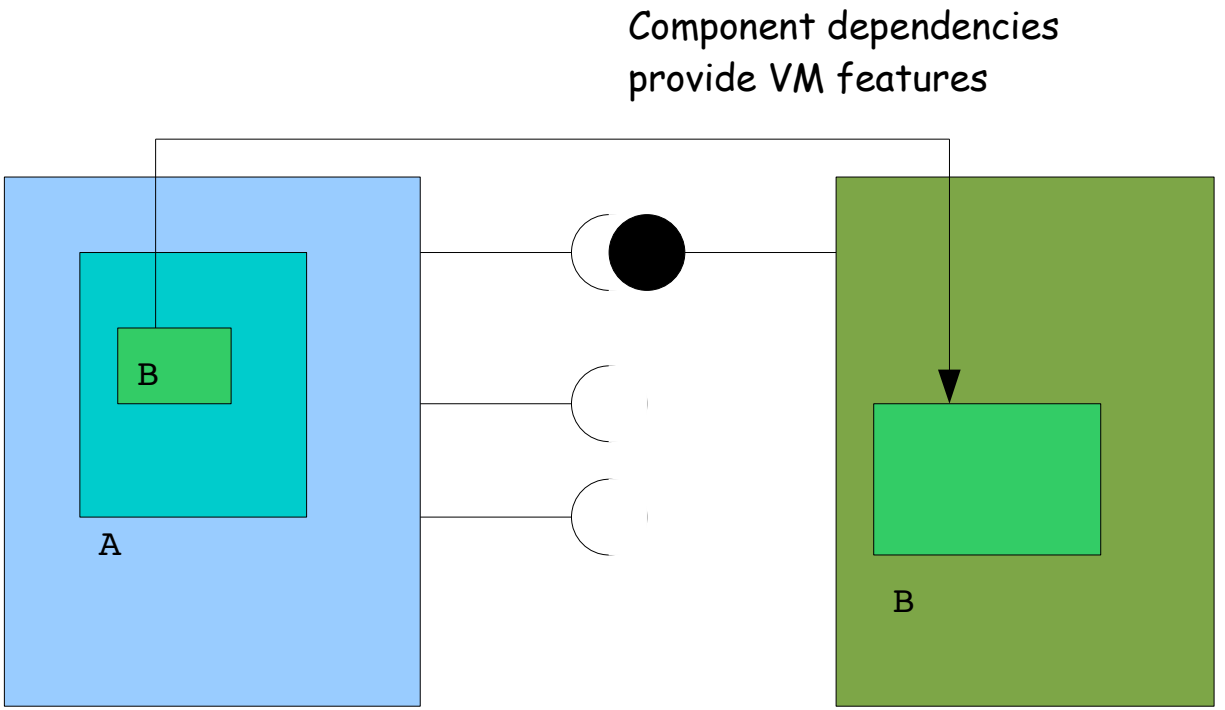
- Tay: a component-based VM
  - Components provide the primitives
  - Load and wire components as required
  - Add and remove features



- The rest of this talk discusses a work in progress to build this architecture

Within a component

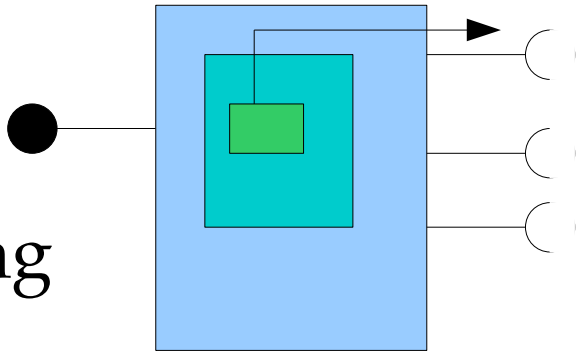
- "Dependencies" = VM operations
  - A reference in one component links to an implementation in another

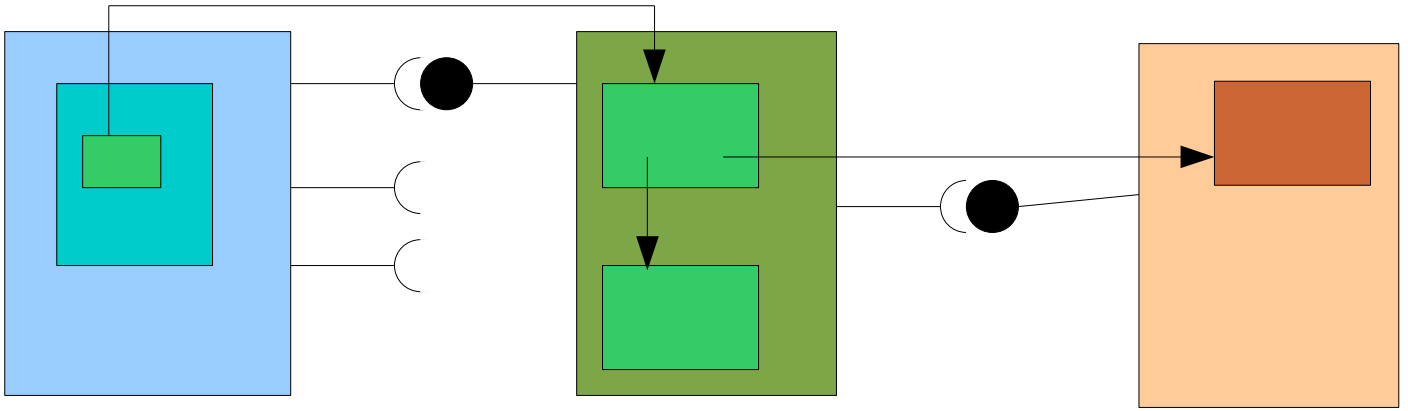


You'd expect features to be provided in a rough hierarchy – although this might not be strict

Within a definition

- "Definition" = VM instruction
  - Analogous to a bytecode
  - Implementation provided by wiring to an implementing component
  - Provide as a primitive
  - Provide using compound definitions in terms of *other* VM instructions





Providing the code

- Bytecode is clearly going to be problematic
  - Not a bit enough namespace
  - ...at least if we consider all the possible features
- Small base of functions to build on
  - As few as possible, to allow specialisation
- Compact code
  - Make sure we can make references small (bytecode would be ideal...)

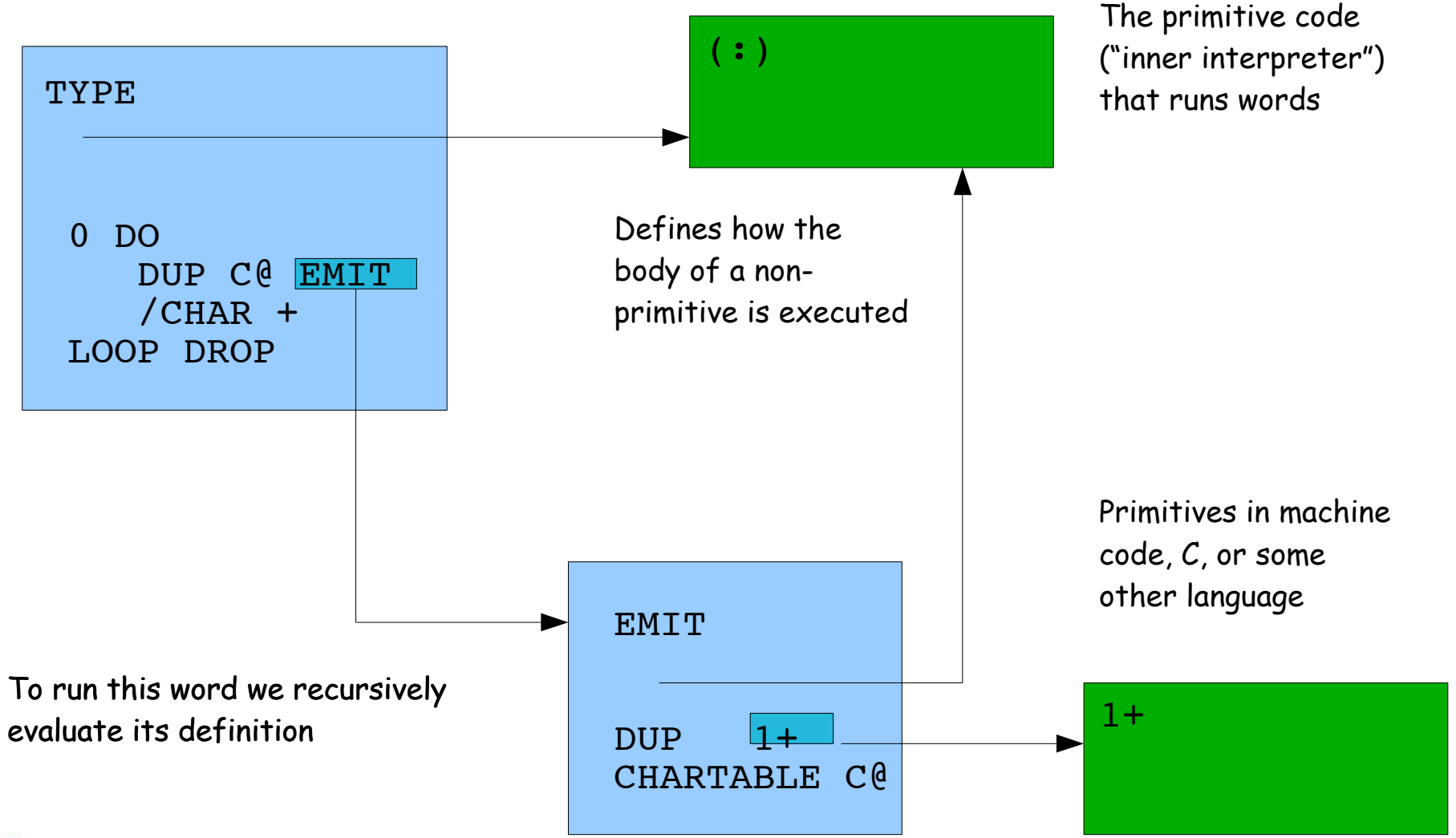
(Questionable) choices

- Monotyped
  - Everything a "cell"
  - Leave typing to higher-level stages/compilers
- Stack machine
  - Fast, small, common, easily ported

- Threaded interpreter
  - Extensible instruction set while retaining (most) compactness

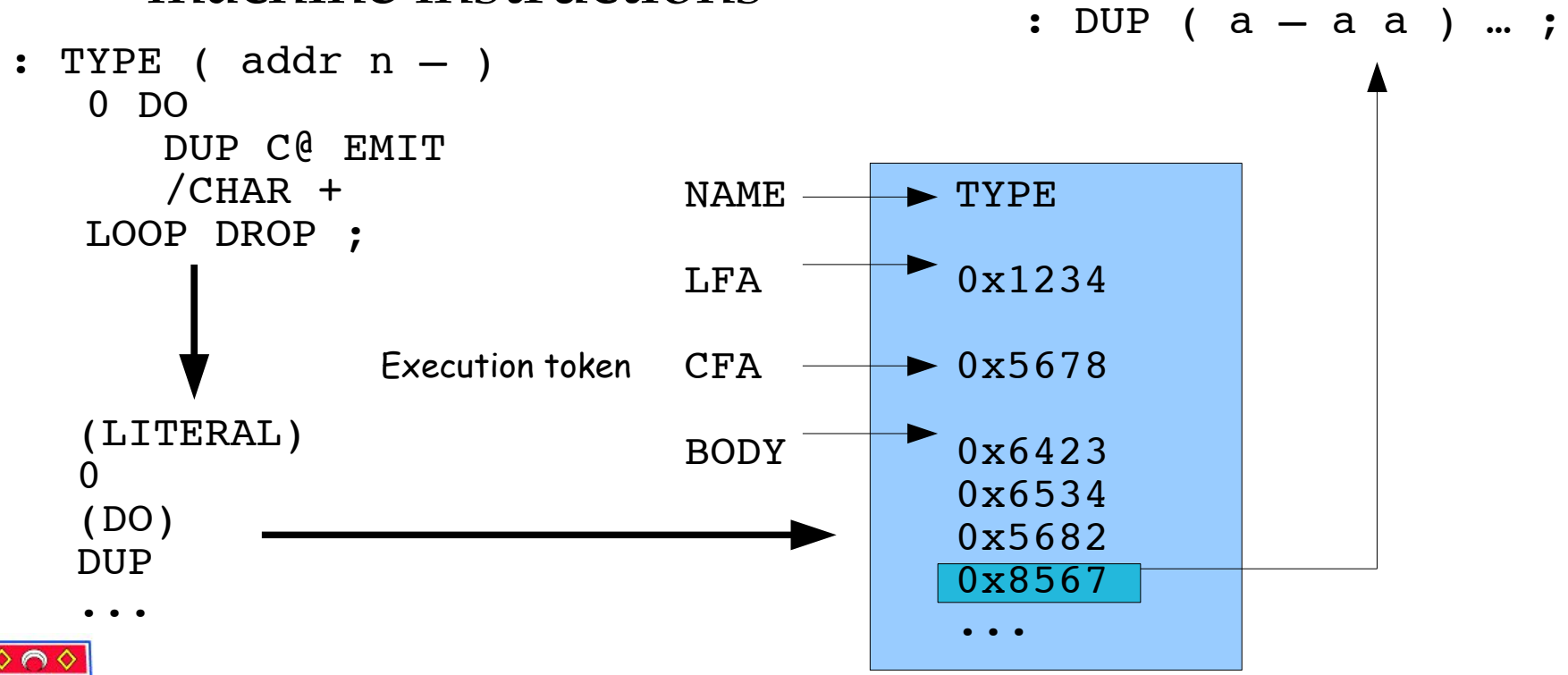
Older viewers may recognise this as the language Forth – and would not be wrong...

Threaded interpretation – 1



Threaded interpretation – 2

- Execution tokens
  - No run-time lookup: a direct jump
  - Word invocation is very cheap, of the order of 10 machine instructions



Threaded VM inner loop

- Decode the execution token and use an "address" to get the primitive code to execute

```
void threaded_inner_interpreter() {
  while(TRUE) {
    xt = *ip++;
    prim = xt_to_behaviour(xt);
    (*prim)(xt);
  }
}
void duplicate( xt ) {
  v = pop(); push(v); push(v);
}
void do_compound( xt ) {
  push_return(ip);
  ip = xt_to_code_body(xt);
}
void end_compound( xt ) {
  ip = pop_return();
}
```

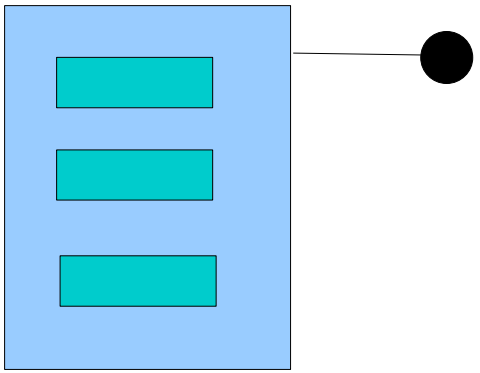
Convert an xt to the behavioural address

Behaviour may be "compound", which pushes the return address onto the VM control stack to later return

The core VM

- We define a component providing the really core concepts
  - Inner interpreter
  - Data and control (return) stack
  - Stack and arithmetic operations
  - Branches and basic control structures

All definitions are primitives, exporting the core VM operations, wiring etc



Requires no other components

Example: loops

```
: TYPE ( addr n — )
  0 DO
      DUP C@ EMIT
      /CHAR +
  LOOP DROP ;
```

Compile the starting behaviour that sets up the stacks etc

```
COUNT
1+
(DO)
DUP
I
+
C@
EMIT
/CHAR
+
(BRANCH) -20
DROP
(NEXT)
```

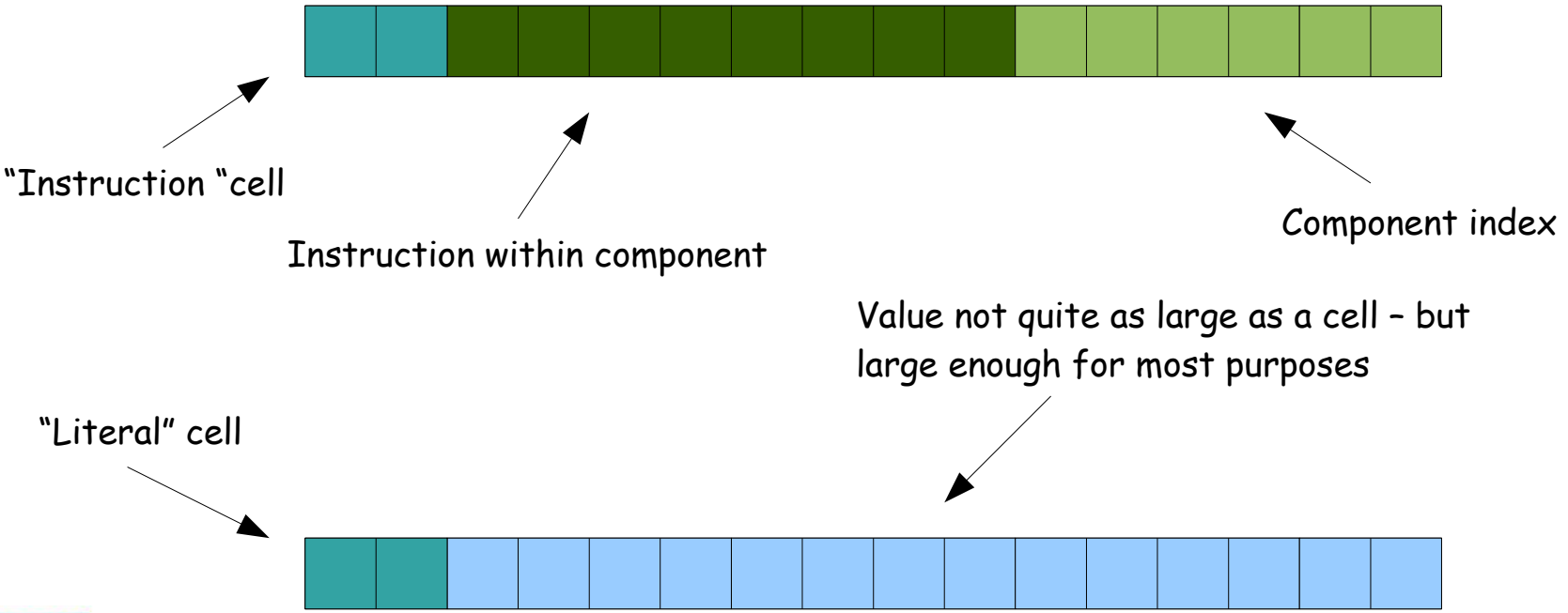
Compile a jump back to the start of the loop

Some words "compile-out" to simpler structures

Cell values – 1

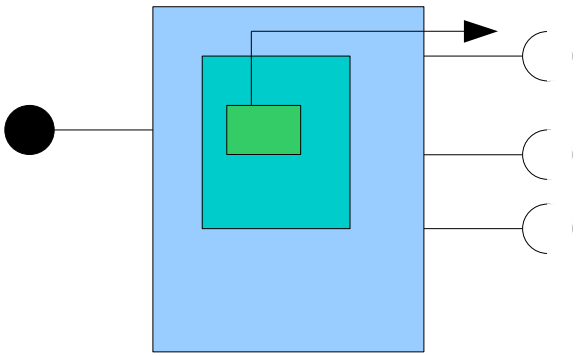
- A reference to something else in the system
  - Instruction, small literal, memory address, …
- Steal bits from a cell

16-bit cells, common for a lot of WSN systems



Cell values – 2

- Because we're treating all operations as instructions, we can abstract an awful lot of things that normally need complex handling
  - Agree on stacks and cells
  - Things like arithmetic are the only way to "unpack" literal cells
  - Parameterise for the number of cell kinds by stealing more bits



A minimal level of agreement seems to be enough to abstract all the other details

Cell value kinds

- Seem to need to define these globally
  - Short integers and reals
  - Instructions
  - Real-memory addresses
  - Heap references (also addresses)
  - Branches
  - Cell sequences landing on the stack

It's a shame this seems to be necessary, but the cell value structure ends up deeply embedded in the inner loop code

These encode enough information to make the heap parseable

Interpretation and compositionality

- Overheads are very small
  - Instruction decode, index and indirect call
  - Around 10 real instructions per VM instruction

- A surprising number of language features are orthogonal
  - Very little needs global agreement
  - Stacks and cells
  - Define and select features as required, don't take the hit for those that aren't needed

Dobson *et alia*. Vanilla: an open language framework. LNCS 1799. 1999.

For example: memory management

- Real memory and managed memory
  - Different cell kinds
  - One component provides static allocation
  - Another provides a heap making use of the static allocator
  - No overhead for systems that don't need a memory manager
  - No way to synthesise managed-memory cells except using the appropriate components

For example: concurrency – 1

- Set up a threaded interpreter that runs a certain number of VM instructions and then returns
  - Force all instructions to be non-blocking
  - VM has an instruction to run a sequence of instructions and then return

```
void bounded_interpreter( bound, thread ) {
  oldthread = activate(thread);
  n = bound;
  while(!prioritised && n--) {
    xt = *ip++;
    prim = xt_to_behaviour(xt);
    (*prim)(xt);
  }
  activate(oldthread);
}
```

Prioritisation allows a thread to hold onto execution temporarily

- One component can then schedule the running of others
  - Lightweight threads

For example: concurrency – 2

- Main thread acts as a scheduler
  - Select another thread to run

Partially run a continuation

- "Surface" concurrency control into the VM
  - Write VM scheduler using all the features available within the VM
  - Round-robin, priorities, interrupts, ...
- Does not have to be built-in or done outside the instruction stream (as with the JVM)
  - Architecturally consistent

Dobson, Porter and Dearle. Bounded first-class control, In preparation.

Choosing the abstraction level

- Changing the VM changes the level of abstraction
  - Program in terms of higher-level constructs
  - Generally leads to more compact code
- Change the abstraction level without re-compiling or -deploying the VM
  - Higher-level operations
  - Target at the operations needed by the WSN

Levis and Culler. Maté: a virtual machine for tiny networked sensors. ACM ASPLOS. 2002.

Does this work?

- Kind of (so far)
  - Some features cross-cut: it makes no sense to use different heap components, but hard to ensure consistent (re-)wiring
  - We've ducked the issue of typing, which can't be addressed using cell kinds, leaving it to compilers that target the Tay VM
  - The *same* xt can refer to *different* instructions because of the interface-relative indexing

Makes debugging interesting

Current state

- We have a basic version working
  - Built using Attila, a from-the-ground-up Forth VM
  - Alter the Attila cross-compiler to build Tay components
  - Basic wiring and other operations

- Will be open-sourced

Moving forward

- Looking towards experimentation
  - Performance seems acceptable
  - Implement a "real" WSN language, for example the InSense language developed in St Andrews

    Balasubramaniam, Dearle and Morrison. A composition-based approach to the construction and dynamic reconfiguration of wireless sensor network applications. LNCS 4954. 2008.

- What are the new language features we need?
  - Easier to explore the space when we only have to provide *new* features, not a complete compiler/VM

Three things to take away

- Un-fix the virtual machine
  - Components all the way down
- Allows elements to be surfaced and re-defined that are normally hard-coded
  - Minimal agreement
- Simplify experimentation with new languages
  - Focus on the new, re-use what we understand