Why Modern Programming Languages Matter

Mark P Jones, Portland State University

Winter 2017

A short history of the automobile

1900 1920 1940 1960 1980 2000 2020

Ford Model T
Ford Model T Pickup (1922)
Utility

1931
Ford Model A Deluxe
Comfort

Volkswagen Type 2 (1949)
Capacity

Fins

1955
Ford Thunderbird
Luxury

Cadillac Eldorado Seville (1959)
Fins

1959
Morris Mini
Compact

Speed

1965
Ford Mustang Coupe
Power

Dodge D200 Camper (1974)
Recreation

1981
DeLorean DMC-12
Time Travel

1989
Ferrari 348
Speed

1997
Toyota Prius
Hybrid

2002
Volkswagen Beetle
Personality

2012
Tesla Model S
Electric

(Images via Wikipedia, subject to Creative Commons and Public Domain licenses)

A short history of programming languages


Lisp
Simula
Smalltalk
BASIC

Fortran
C
Pascal

An early systems programming language, sometimes described as “portable assembler”

(Images via Wikipedia, subject to Creative Commons and Public Domain licenses)
C is great … what more could you want?

- Programming in C gives systems developers:
  - Good (usually predictable) performance characteristics
  - Low-level access to hardware when needed
  - A familiar and well-established notation for writing imperative programs that will get the job done

- What can you do in modern languages that you can’t already do with C?

- Do you really need the fancy features of newer object-oriented or functional languages?

- Are there any downsides to programming in C?

Could a different language make it impossible to write programs with errors like these?

Impact: An application may be able to execute arbitrary code with kernel privileges
Description: Multiple memory corruption issues were addressed through improved input validation.

Impact: An application may be able to execute arbitrary code
Description: A use after free issue was addressed through improved input validation.

Impact: A local user may be able to gain root privileges
Description: A type confusion issue was addressed through improved memory handling.

Impact: An application may be able to execute arbitrary code
Description: An out-of-bounds write issue was addressed by removing the vulnerable code.
The Habit programming language

- "a dialect of Haskell that is designed to meet the needs of high assurance systems programming"

  **Habit** = **Haskell + bits**

- Habit, like Haskell, is a functional programming language
- For people trained in using C, the syntax and features of Habit may be unfamiliar
- I won’t assume familiarity with functional programming here
- We’ll focus on how Habit uses types to detect and prevent common types of programming errors

Division

- You can divide an integer by an integer to get an integer result
- In Habit: `div :: Int → Int → Int`

  - This is a lie!
  - **Correction:** You can divide an integer by a **non-zero integer** to get an integer result

  - In Habit:

    ```
    div :: Int → NonZero Int → Int
    ```

  - But where do **NonZero Int** values come from?

Where do **NonZero** values come from?

- **Option 1**: Integer literals - numbers like 1, 7, 63, and 128 are clearly all **NonZero** integers

- **Option 2**: By checking at runtime

  ```
  nonzero :: Int → Maybe (NonZero Int)
  ```

  Values of type **Maybe t** are either:
  - **Nothing**
  - **Just x for some x of type t**

  - These are the only two ways to get a **NonZero Int**!
  - **NonZero** is an abstract datatype

Examples using **NonZero** values

- Calculating the average of two values:

  ```
  ave :: Int → Int → Int
  ave n m = (n + m) `div` 2
  ```

- Calculating the average of a list of integers:

  ```
  average :: List Int → Maybe Int
  average nums
  = case nonzero (length nums) of
    Just d → Just (sum nums `div` d)
    Nothing → Nothing
  ```

  - Key point: If you forget the check, your code will not compile!
Null pointer dereferences

- In C, a value of type $T^*$ is a pointer to an object of type $T$
- But this may be a lie!
- A null pointer has type $T^*$, but does NOT point to an object of type $T$
- Attempting to read or write the value pointed to by a null pointer is called a “null pointer dereference” and often results in system crashes, vulnerabilities, or memory corruption
- Described by Tony Hoare (who introduced null pointers in the ALGOL W language in 1965) as his “billion dollar mistake”

Pointers and reference types

- Lesson learned: we should distinguish between
  - References (of type $\text{Ref } a$): guaranteed to point to values of type $a$
  - Pointers (of type $\text{Ptr } a$): either a reference or a null
- These types are not the same: $\text{Ptr } a = \text{Maybe } (\text{Ref } a)$
- You can only read or write values via a reference
- Code that tries to read from a pointer will fail to compile!
- Goodbye null pointer dereferences!

Arrays and out of bounds indexes:

- Arrays are collections of values stored in contiguous locations in memory
  - Address of $a[i] = \text{start address of } a + i \times \text{(size of element)}$
  - Simple, fast, … and dangerous!
  - If $i$ is not a valid index (an “out of bounds index”), then an attempt to access $a[i]$ could lead to a system crash, memory corruption, buffer overflows, …
  - A common path to “arbitrary code execution”

Array bounds checking

- The designers of C knew that this was a potential problem … but chose not to address it in the language design:
  - We would need to store a length field in every array
  - We would need to check for valid indexes at runtime
- The designers of Java knew that this was a potential problem … and chose to address it in the language design:
  - Store a length field in every array
  - Check for valid indexes at runtime
- Performance **OR** Safety … pick one!
Arrays in Habit

• Key idea: make array size part of the array type, do not allow arbitrary indexes:

\[(@) :: \text{Ref} \ (\text{Array} \ n \ t) \rightarrow \text{Ix} \ n \rightarrow \text{Ref} \ t\]

\(a[i]\) is written as \(a@i\) in Habit

• Fast, no need for a runtime check, no need for a stored length

• \(\text{Ix} \ n\) is another abstract type:

\[\begin{align*}
\text{maybeIx} &:: \text{Int} \rightarrow \text{Maybe} \ (\text{Ix} \ n) \\
\text{modIx} &:: \text{Int} \rightarrow \text{Ix} \ n \\
\text{incIx} &:: \text{Ix} \ n \rightarrow \text{Maybe} \ (\text{Ix} \ n)
\end{align*}\]

In assembly

\[\begin{align*}
\text{movl} &\quad \text{base}, \%eax \\
\text{movl} &\quad \text{limit}, \%ebx \\
\text{mov} &\quad \%eax, \%edx \\
\text{shl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%edx, \%ecx \\
\text{shr} &\quad \%edx, \%edx \\
\text{xor} &\quad \%ecx, \%edx \\
\text{shl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%edx, \%ecx \\
\text{and} &\quad \%ecx, \%edx \\
\text{orl} &\quad \%edx, \%edx \\
\text{orl} &\quad \%edx, \%edx \\
\text{movl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%eax, \%edx \\
\text{mov} &\quad \%edx, \%edx \\
\text{shl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%edx, \%edx \\
\text{and} &\quad \%edx, \%edx \\
\text{or} &\quad \%edx, \%edx \\
\text{orl} &\quad \%edx, \%edx \\
\text{movl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%eax, \%edx \\
\text{mov} &\quad \%edx, \%edx \\
\text{shl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%edx, \%edx \\
\text{and} &\quad \%edx, \%edx \\
\text{or} &\quad \%edx, \%edx \\
\text{orl} &\quad \%edx, \%edx \\
\text{movl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%eax, \%edx \\
\text{mov} &\quad \%edx, \%edx \\
\text{shl} &\quad \%edx, \%edx \\
\text{mov} &\quad \%edx, \%edx \\
\text{and} &\quad \%edx, \%edx \\
\text{or} &\quad \%edx, \%edx \\
\text{orl} &\quad \%edx, \%edx
\end{align*}\]

In C

\[\begin{align*}
\text{low} & \quad = \ (\text{base} \ll 16) \quad \quad \quad // \text{purple} \\
\quad & \quad \quad | \ (\text{limit} \ & \ 0xffff); \quad \quad // \text{blue} \\
\text{high} & \quad = \ (\text{base} \ & \ 0xffff000000) \quad // \text{pink} \\
\quad & \quad \quad | \ (\text{limit} \ & \ 0xf0000) \quad // \text{green} \\
\quad & \quad \quad | \ ((\text{base} \gg 16) \ & \ 0xff) \quad // \text{yellow} \\
\quad & \quad \quad \quad | \ 0x503200; \quad \quad // \text{white}
\end{align*}\]

• Examples like this show why we use high-level languages instead of assembly!

• But let's hope we don't get those offsets and masks wrong …

• And there is no safety net if we get the types wrong …
In Habit

• Programmer describes layout in a type definition:

```
bitdata GDT
  = GDT [ pink  :: Bit 8 | 0x5  :: Bit 4
          | green :: Bit 4 | 0x32 :: Bit 8
          | yellow :: Bit 8 | purple, blue :: Bit 16 ]
```

• Compiler tracks types and automatically figures out appropriate offsets and masks:

```
makeGDT :: Unsigned → Unsigned → GDT
makeGDT (pink # yellow # purple) -- base
       (0 # green # blue)       -- limit
  = GDT [pink|green|yellow|purple|blue]

silly :: GDT → Bit 8
silly gdt = gdt.pink + gdt.yellow
```

Additional examples

• Layout and initialization of memory-based tables and data structures
• Distinguishing physical and virtual addresses
• Tracking (and limiting) side effects
  • ensuring some sections of code are “read only”
  • identifying/limiting code that uses privileged operations
  • preventing code that sleeps while holding a lock
    • …
• Reusable methods for concise and consistent input validation…
  • …
The CEMLaBS Project

- Three **technical** questions:
  - **Feasibility**: Is it possible to build an inherently “unsafe” system like seL4 in a “safe” language like Habit?
  - **Benefit**: What benefits might this have, for example, in reducing development or verification costs?
  - **Performance**: Is it possible to meet reasonable performance goals for this kind of system?

- A **social** question:
  - Can we persuade developers to try new languages?
  - Maybe there is a role for modern programming languages …!?