NICTA

CDSL

A Restricted Functional Language for File System Verification

Liam O'Connor

FP-Syd, October 2013

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

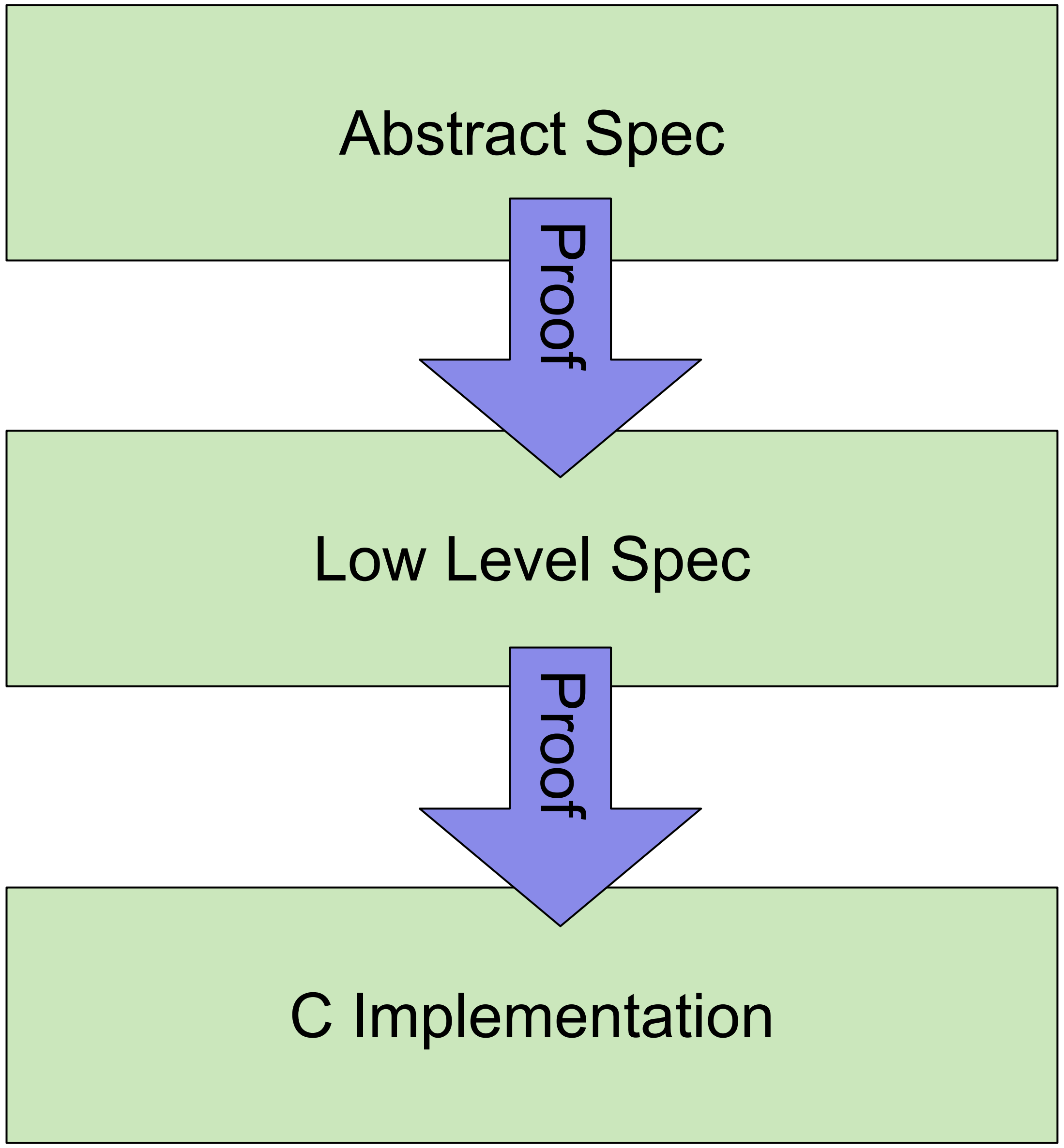
seL4

Abstract Spec

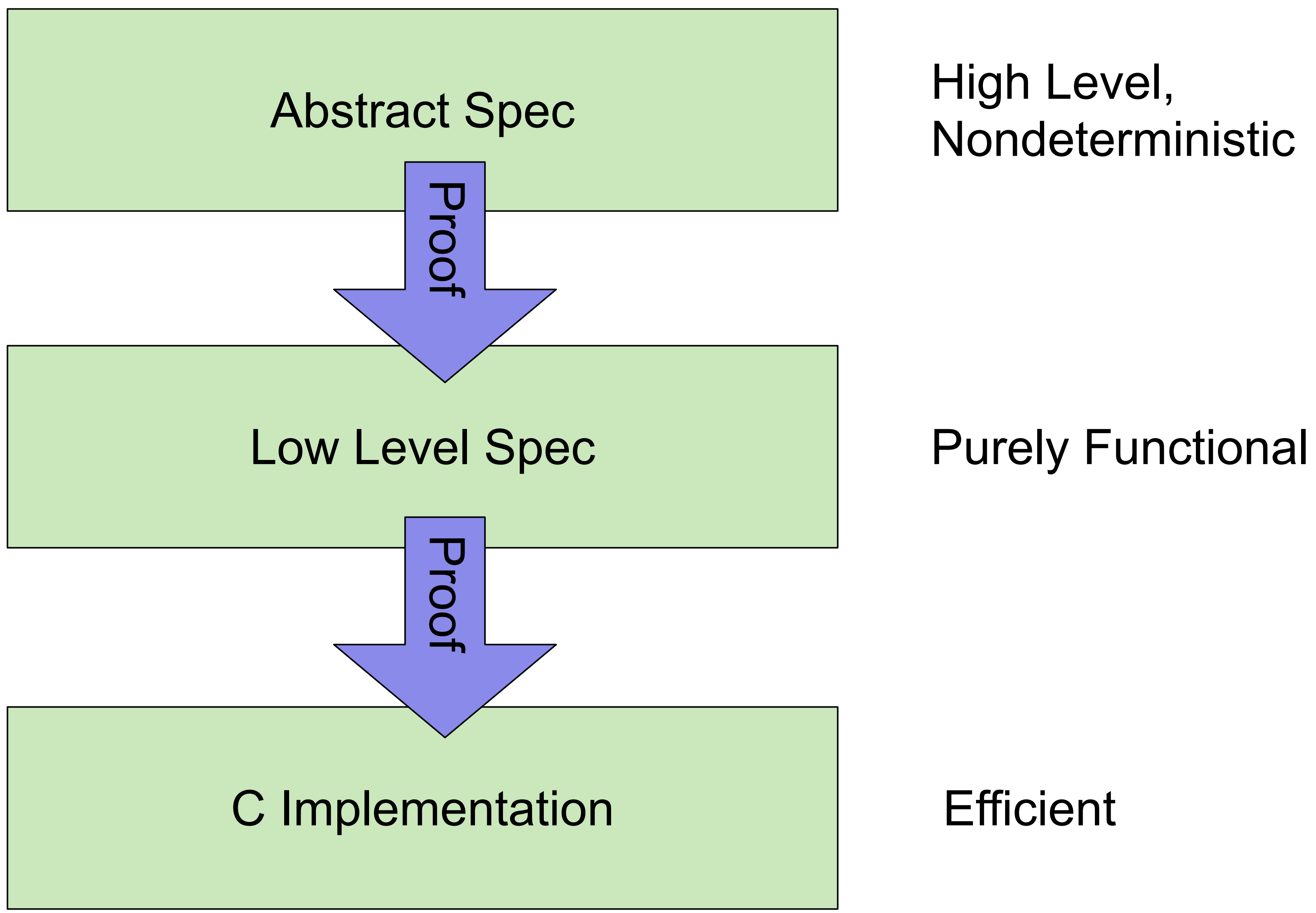
Low Level Spec

C Implementation

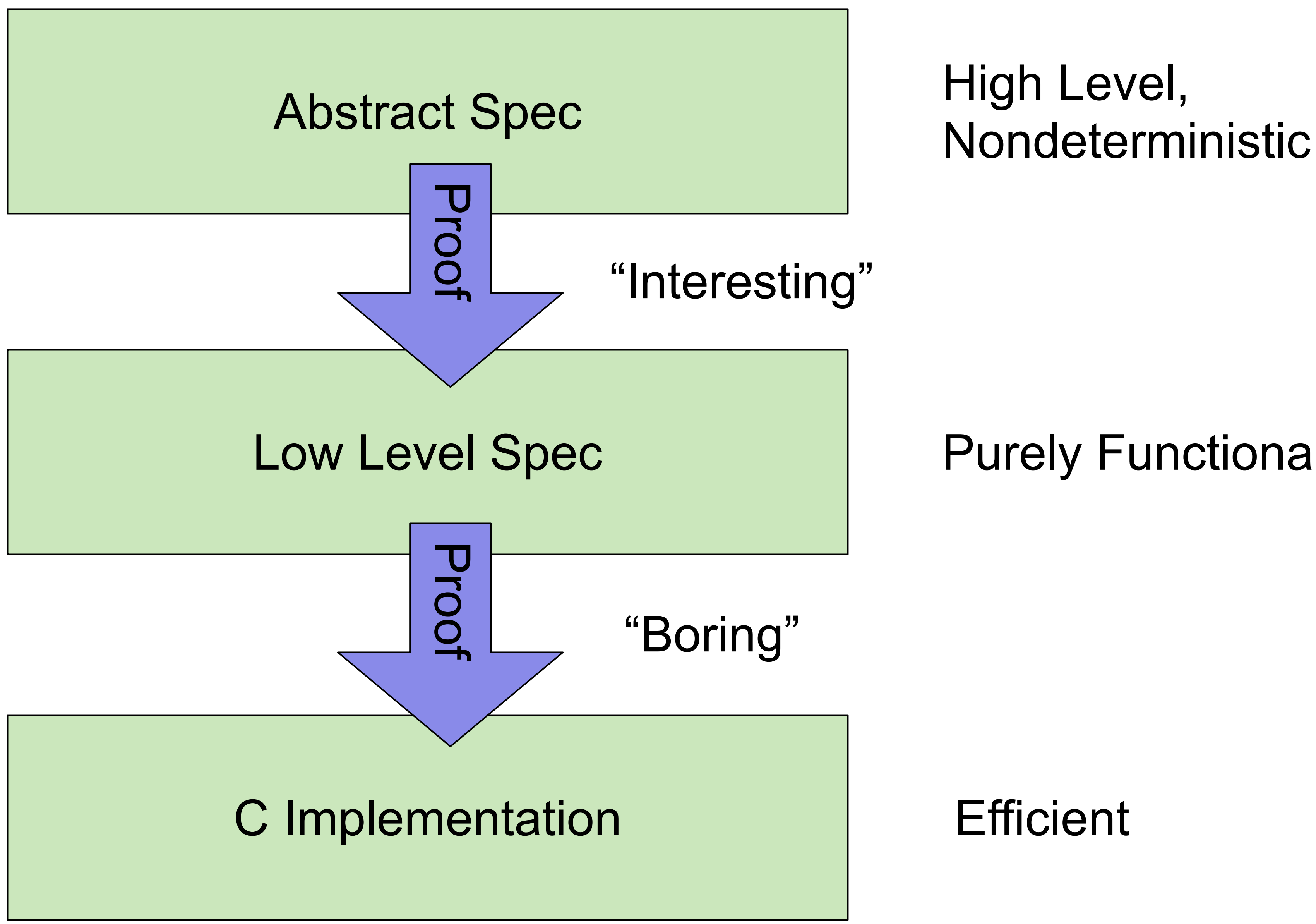
seL4



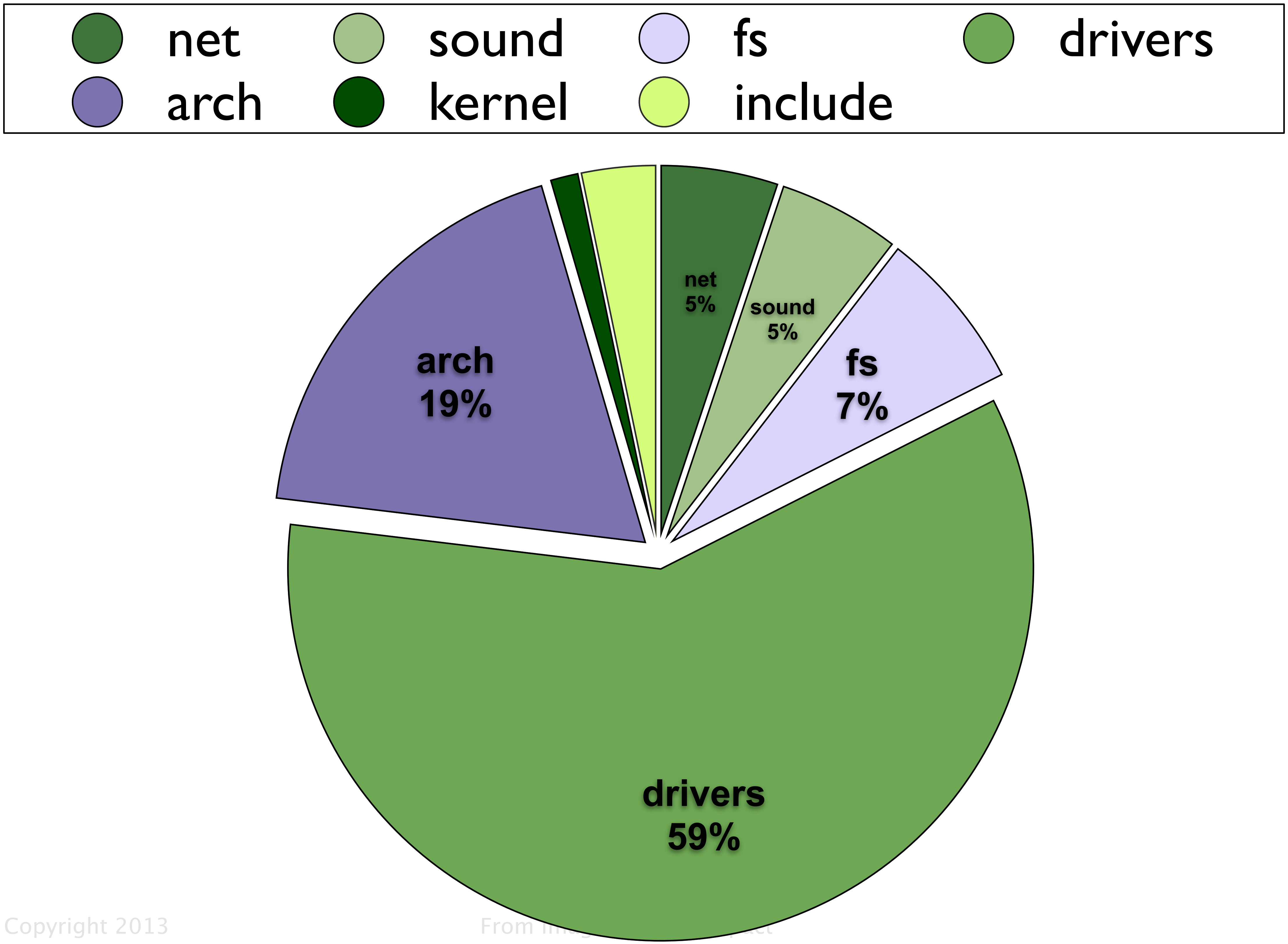
seL4


Abstract Spec
Proof
Low Level Spec
Proof
C Implementation
High Level,
Nondeterministic
Purely Functional
Efficient

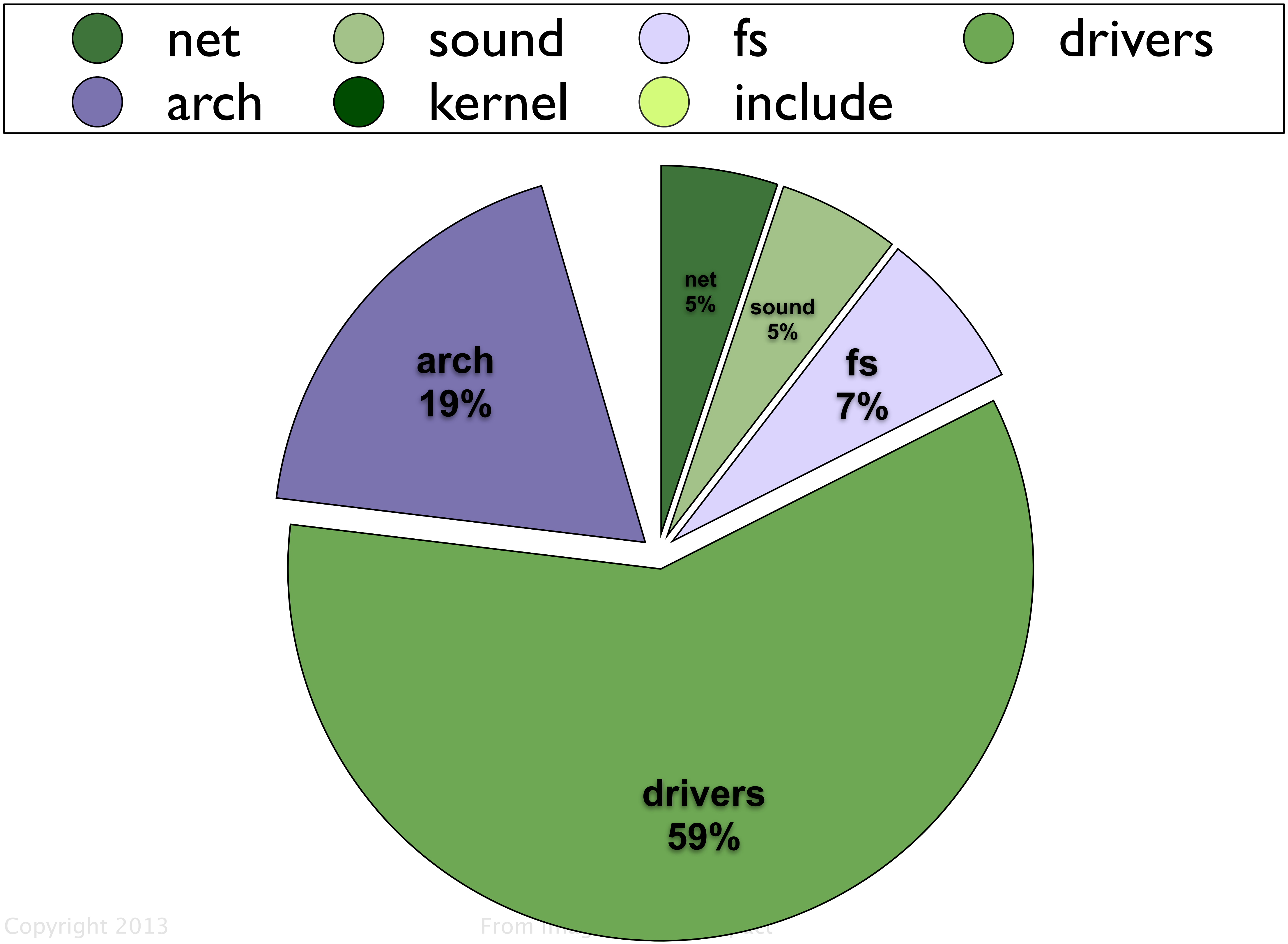
seL4


Abstract Spec
Proof
Low Level Spec
Proof
C Implementation
High Level,
Nondeterministic
"Interesting"
Purely Functional
"Boring"
Efficient

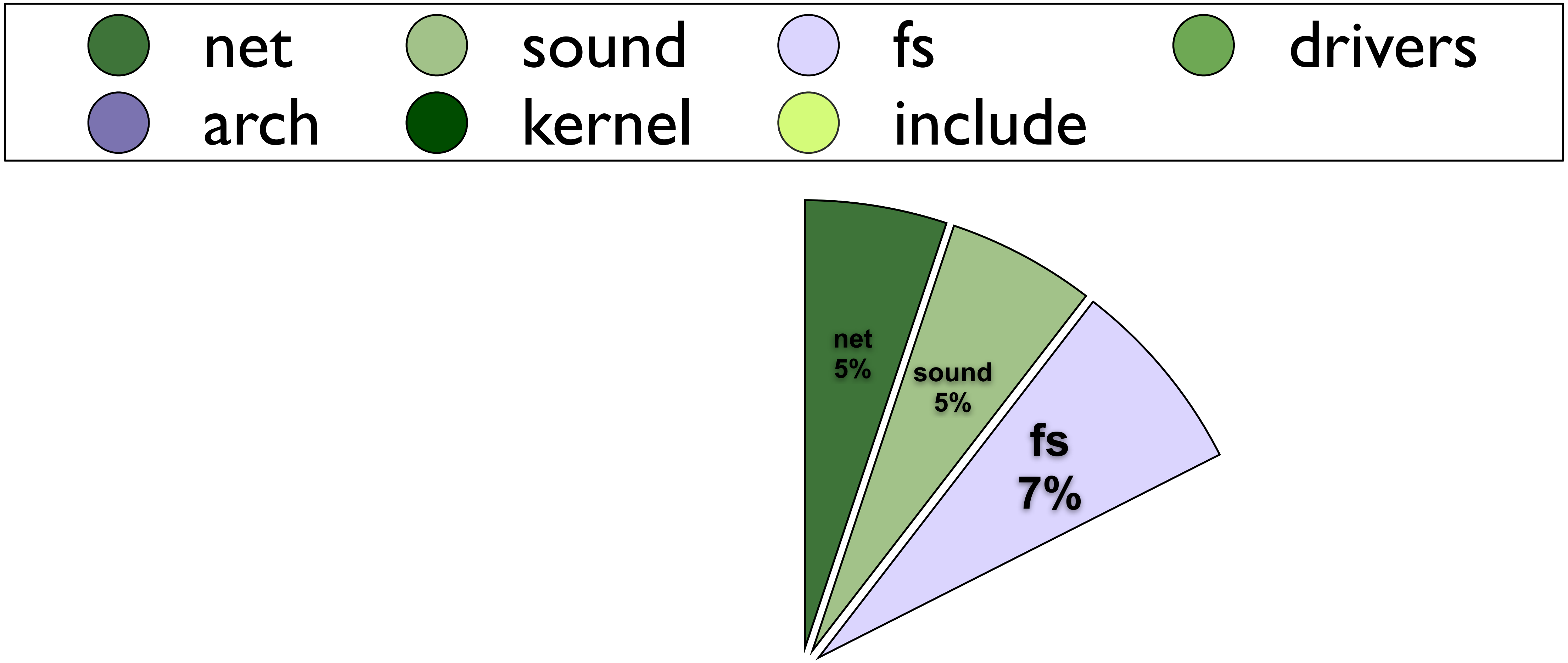
Typical OS


net
sound
fs
drivers
arch
kernel
include
net 5%
sound 5%
fs 7%
arch 19%
drivers 59%

Typical OS


net
sound
fs
drivers
arch
kernel
include
arch 19%
net 5%
sound 5%
fs 7%
drivers 59%

Typical OS



Economy of Scale

```
liamoc@duvel:~$ cat /proc/filesystems | wc -l
31

liamoc@tstvm:~$ cat /proc/filesystems  | wc -l
49
```


Economy of Scale

```
liamoc@duvel:~$ cat /proc/filesystems | wc -l
31

liamoc@tstvm:~$ cat /proc/filesystems  | wc -l
49
```

We don't want a cathedral, we want a factory!

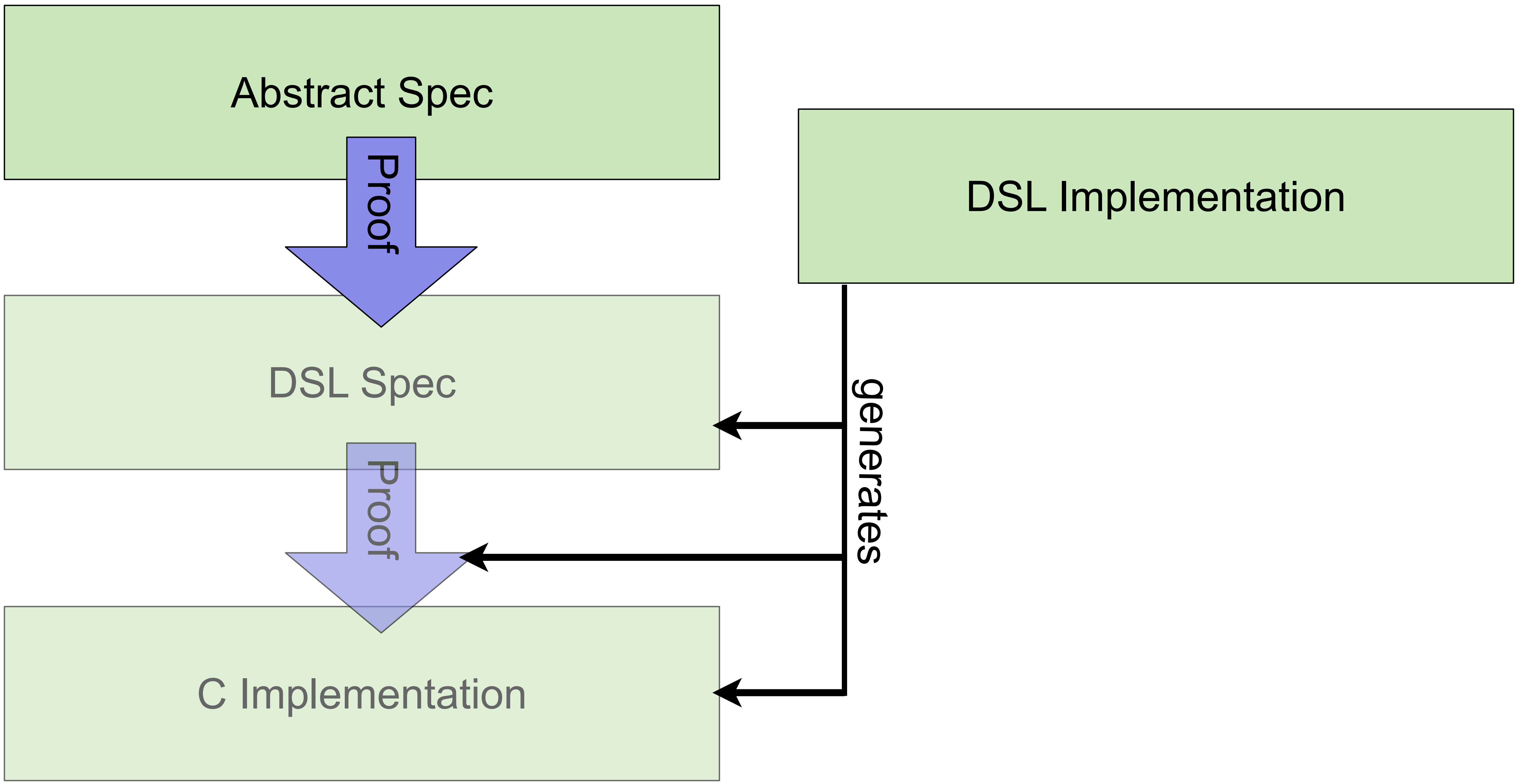
Economy of Scale

```
liamoc@duvel:~$ cat /proc/filesystems | wc -l
31

liamoc@tstvm:~$ cat /proc/filesystems  | wc -l
49
```

We don't want a cathedral, we want a ~~factory!~~ DSL!

Our Approach



Wishlist

Wishlist

- Our DSL needs to:
  - Establish key verification properties:
    - Type/Memory Safety, Termination, Totality
  - Compile to efficient C code
    - Destructive updates, resource disposal, no excessive copying, etc.
  - Be capable of expressing code for FS operations
    - Create file, rename file, etc.

Wishlist

- Our DSL needs to:
  - Establish key verification properties:
    - Type/Memory Safety, Termination, Totality
  - Compile to efficient C code
    - Destructive updates, resource disposal, no excessive copying, etc.
  - Be capable of expressing code for FS operations
    - Create file, rename file, etc.

- We do NOT need to express everything in DSL
  - Can use abstraction
  - Define once, verify once (manually)
  - These components should be used in every file system

Simply typed λ-calculus

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash t : \rho}{\Gamma \vdash \lambda(x :: \tau).\ t : \tau \to \rho}$$

$$\frac{\Gamma \vdash a : \tau \to \rho \quad \Gamma \vdash b : \tau}{\Gamma \vdash a\ b : \rho}$$

Simply typed λ-calculus

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash t : \rho}{\Gamma \vdash \lambda(x :: \tau).\ t : \tau \to \rho}$$

$$\frac{\Gamma \vdash a : \tau \to \rho \quad \Gamma \vdash b : \tau}{\Gamma \vdash a\ b : \rho}$$

$$\frac{\Gamma_1 \Gamma_2 \vdash P}{\Gamma_2 \Gamma_1 \vdash P} \qquad \frac{x : \tau, x : \tau, \Gamma \vdash P}{x : \tau, \Gamma \vdash P} \qquad \frac{\Gamma \vdash P}{x : \tau, \Gamma \vdash P}$$

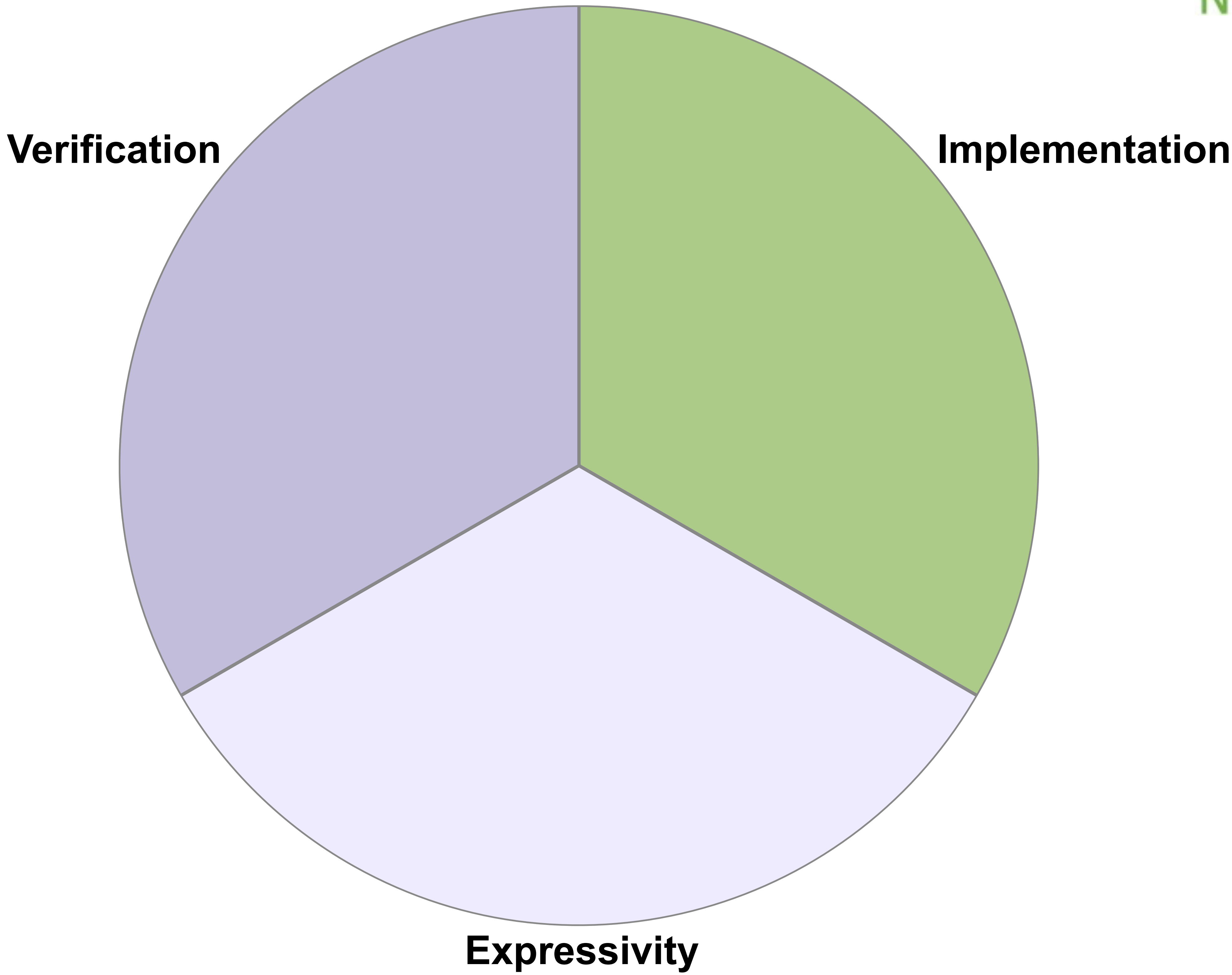
First Order Language

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma, x : \tau \vdash t' : \rho}{\Gamma \vdash \mathtt{let}\ x :: \tau = t\ \mathtt{in}\ t' : \rho}$$

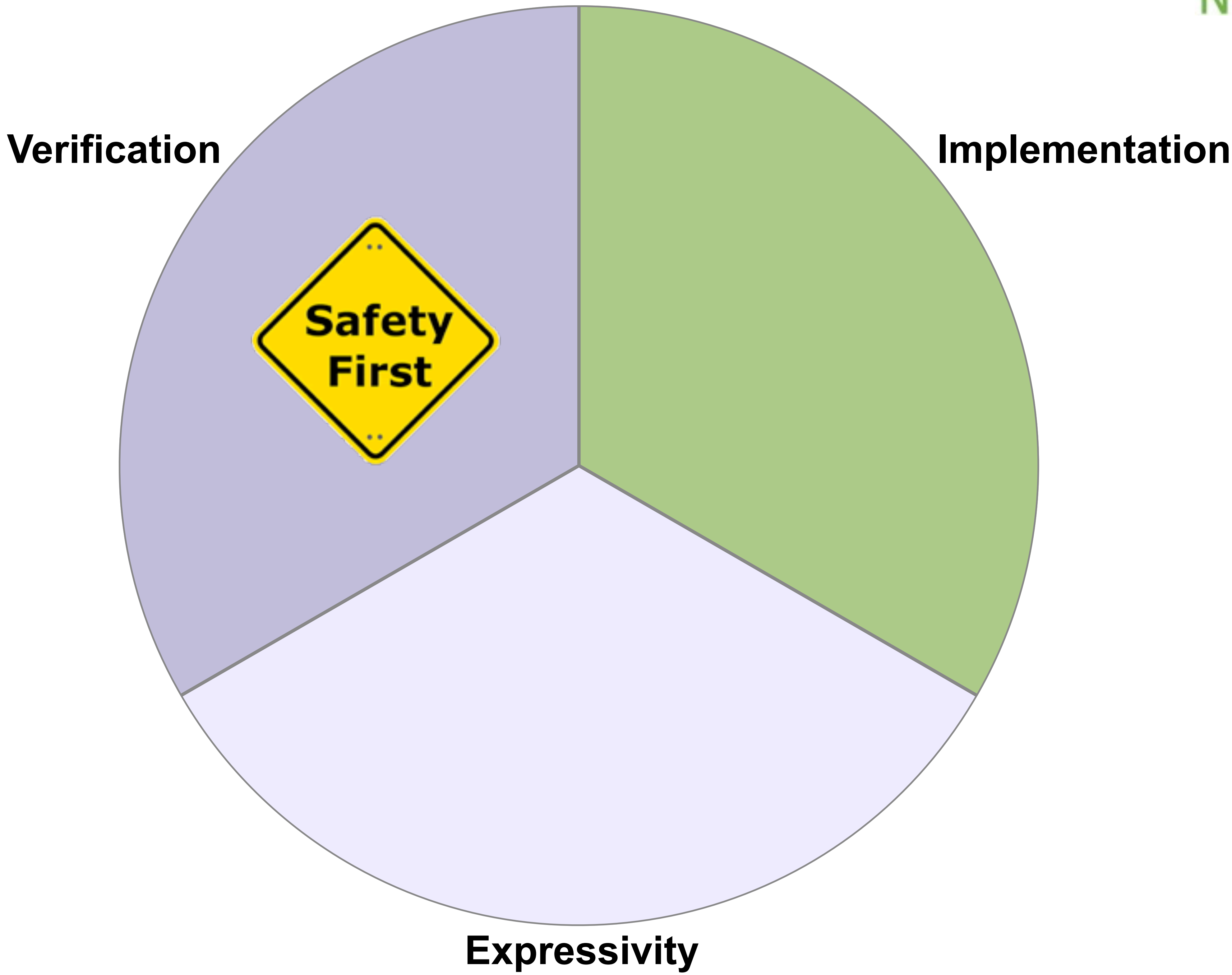
$$\frac{\Gamma \vdash f : \tau \to \rho \quad \Gamma \vdash x : \tau}{\Gamma \vdash f(x) : \rho}$$

$$\frac{\Gamma_1 \Gamma_2 \vdash P}{\Gamma_2 \Gamma_1 \vdash P} \qquad \frac{x : \tau, x : \tau, \Gamma \vdash P}{x : \tau, \Gamma \vdash P} \qquad \frac{\Gamma \vdash P}{x : \tau, \Gamma \vdash P}$$

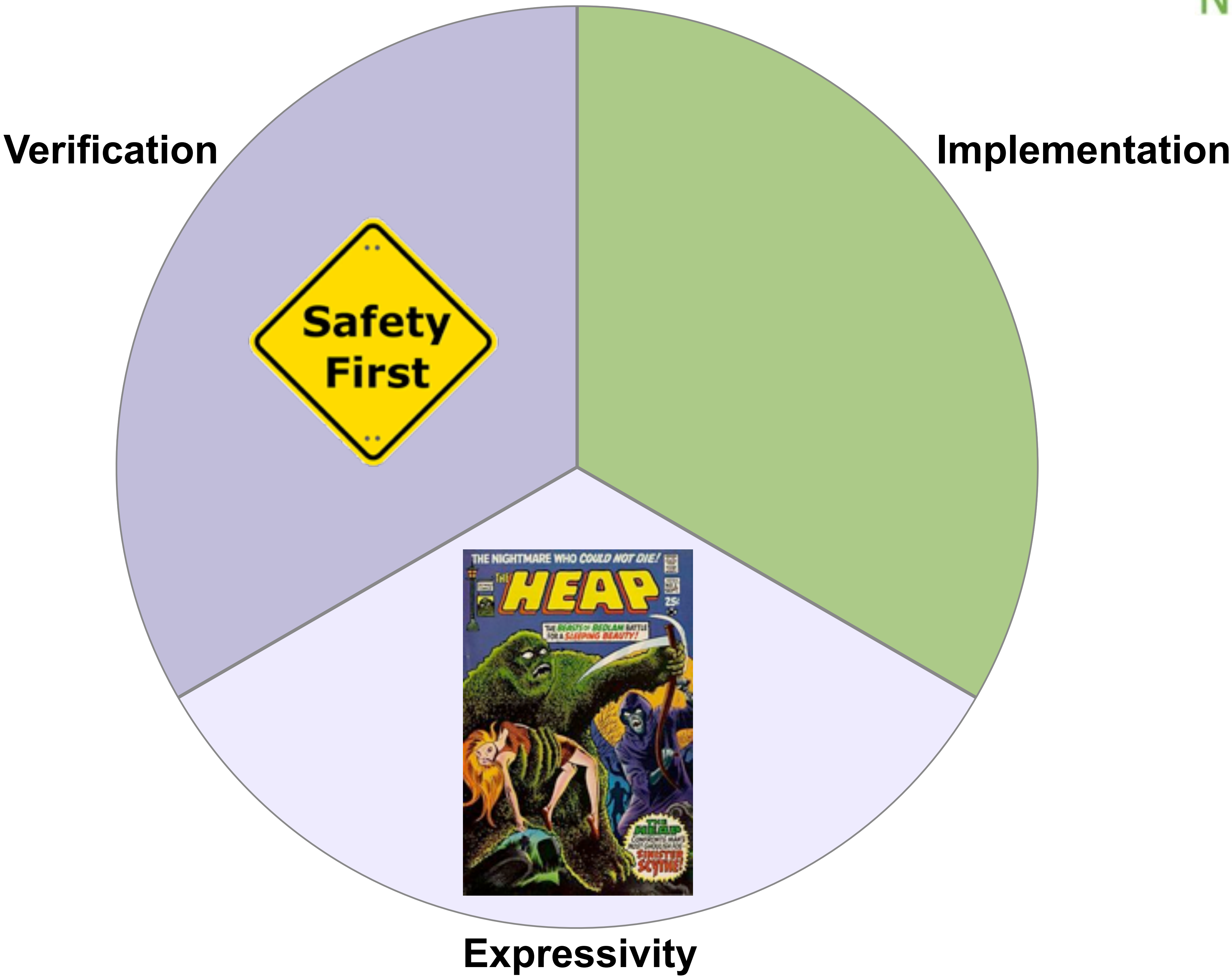
Memory Management



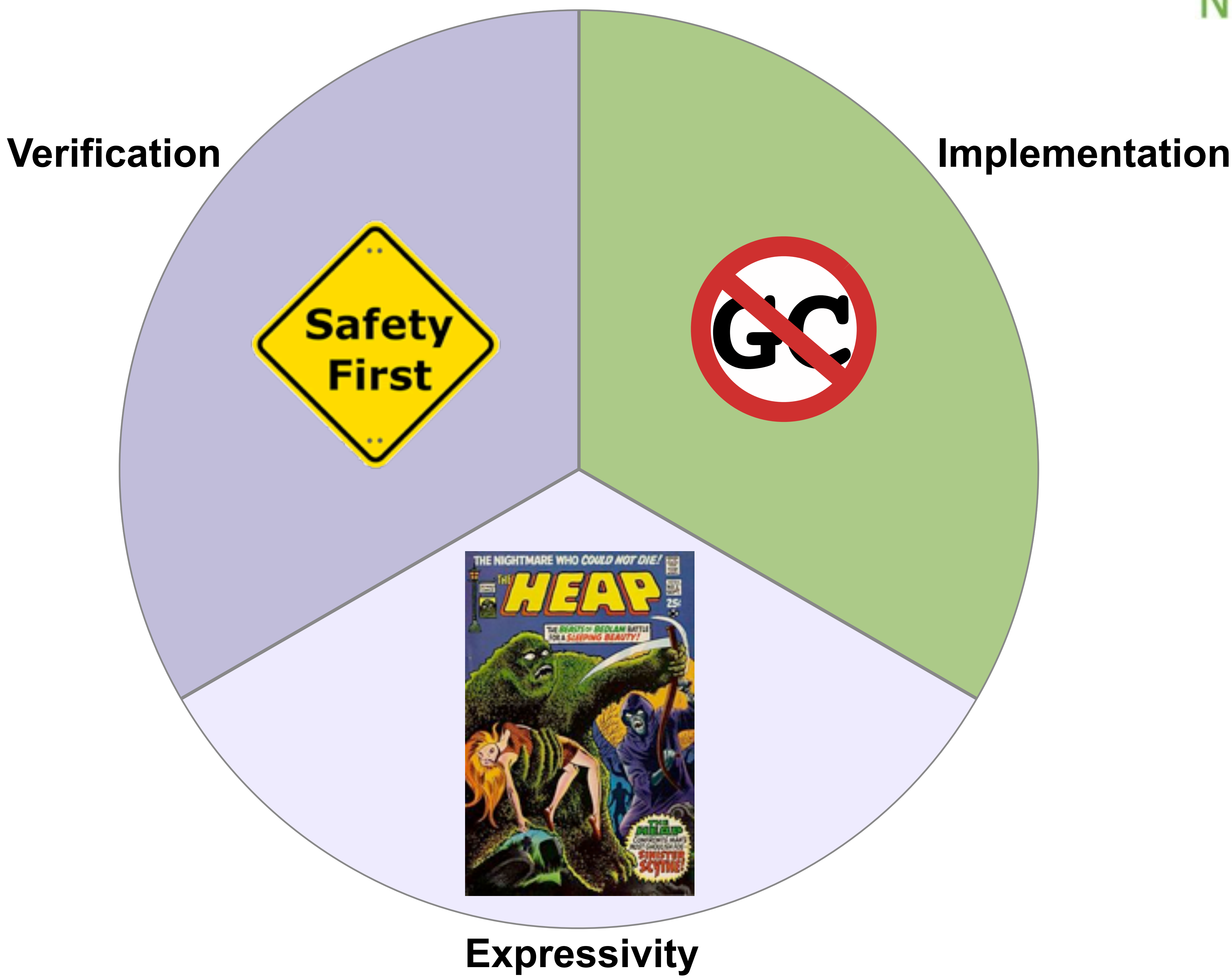
Memory Management



Memory Management



Memory Management



Memory Management

Memory Management

- Automatic memory management (GC) is too big a burden
  - Many static auto-MM techniques are also either inefficient or unsafe

Memory Management

- Automatic memory management (GC) is too big a burden
  - Many static auto-MM techniques are also either inefficient or unsafe

- But what about manual memory management?
  - ```
    let x  = allocData ()
        x' = updateData x
        _  = free x
    in x'
    ```
    ```

Memory Management

- Automatic memory management (GC) is too big a burden
  - Many static auto-MM techniques are also either inefficient or unsafe

- But what about manual memory management?

```
- let x  = allocData ()
      x' = updateData x
      _  = free x
  in x'
```

- But that's unsafe/inefficient/terrible!
  - Types to the rescue!

Linear, First Order Language

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{\Gamma_1 \vdash t : \tau \quad \Gamma_2, x : \tau \vdash t' : \rho}{\Gamma_1 \Gamma_2 \vdash \mathtt{let}\ x :: \tau = t\ \mathtt{in}\ t' : \rho}$$

$$\frac{\Gamma_1 \vdash f : \tau \to \rho \quad \Gamma_2 \vdash x : \tau}{\Gamma_1 \Gamma_2 \vdash f(x) : \rho}$$

$$\frac{\Gamma_1 \Gamma_2 \vdash P}{\Gamma_2 \Gamma_1 \vdash P} \qquad \cancel{\frac{x : \tau, x : \tau, \Gamma \vdash P}{x : \tau, \Gamma \vdash P}} \qquad \cancel{\frac{\Gamma \vdash P}{x : \tau, \Gamma \vdash P}}$$

Safety!?

```
- let x = allocData
      y = x
      _ = free x
  in y
```

$$\cancel{\frac{x : \tau, x : \tau, \Gamma \vdash P}{x : \tau, \Gamma \vdash P}}$$

```
- let y = allocData
  in ()
```

$$\cancel{\frac{\Gamma \vdash P}{x : \tau, \Gamma \vdash P}}$$

```
- let x  = allocData ()
      x' = updateData x
      _  = free x
  in x'
```

$$\cancel{\frac{x : \tau, x : \tau, \Gamma \vdash P}{x : \tau, \Gamma \vdash P}}$$

Note: CDSL core syntax, not surface syntax.

Safety!?

Safety!?

- This example seems safe, but rejected by linear types:
  - ```
    let x  = allocData ()
        x' = updateData x
        _  = free x
      in x'
    ```
    ```

Safety!?

- This example seems safe, but rejected by linear types:
  - ```
    let x  = allocData ()
        x' = updateData x
        _  = free x
     in x'
    ```
- But, `updateData` is not expressible!
  - It has to free `x` or it would be using dereliction
  - It could destructively update it...
    ```

Safety!?

- This example seems safe, but rejected by linear types:
  - ```
    let x  = allocData ()
        x’ = updateData x
        _  = free x
      in x’
    ```
- But, `updateData` is not expressible!
  - It has to free `x` or it would be using dereliction
  - It could destructively update it...
  - ```
    let x  = allocData ()
        x’ = updateData! x
      in x’
    ```

Two Semantics

Two Semantics

- Value Semantics
  - Imagine everything is passed by value
  - There is no heap (free is a no-op)
  - Everything is immutable
  - **Great for reasoning**

Two Semantics

- Value Semantics
  - Imagine everything is passed by value
  - There is no heap (free is a no-op)
  - Everything is immutable
  - **Great for reasoning**
- Update Semantics
  - Some things are stored in the heap
  - Destructive updates actually overwrite memory
  - Free actually deallocates memory
  - **Great for implementation**

Two Semantics

- Value Semantics
  - Imagine everything is passed by value
  - There is no heap (free is a no-op)
  - Everything is immutable
  - **Great for reasoning**
- Update Semantics
  - Some things are stored in the heap
  - Destructive updates actually overwrite memory
  - Free actually deallocates memory
  - **Great for implementation**

Linear Types allow for both views!

Unboxed types

- Some things we *do* want passed by value
  - Unboxed types, integers, small structs, etc.
  - They shouldn't be linear!
  - Functions shouldn't be linear either, or we could only call them once.

  - **Simple solution:**
    - allow dereliction and contraction for certain types.

$T_\bullet$ $T_\sharp$

Case study: Buffer interface

```
- make : () -> .Buf
- free : .Buf -> ()
- length : .Buf -> (#U32, .Buf)



- serialise : (.Obj, .Buf) -> (.Obj, .Buf)

- deserialise : .Buf -> (.Obj, .Buf)
```

Case study: Buffer interface

```
- make : () -> .Buf
- free : .Buf -> ()
- length : .Buf -> (#U32, .Buf)
```

Same!

```
- serialise : (.Obj, .Buf) -> (.Obj, .Buf)

- deserialise : .Buf -> (.Obj, .Buf)
```

Case study: Buffer interface

```
- make : () -> .Buf
- free : .Buf -> ()
- length : .Buf -> (#U32, .Buf)
```

Same!

```
- serialise : (.Obj, .Buf) -> (.Obj, .Buf)

- deserialise : .Buf -> (.Obj, .Buf)
```

Same!

Shareable Values

- We need (nonlinear) “look, don’t touch” references.
  - `make : () -> .Buf`
  - `free : .Buf -> ()`
  - `length : *Buf -> #U32`
  - `serialise : (*Obj, .Buf) -> .Buf`
  - `deserialise : *Buf -> .Obj`

$$\frac{\Gamma_1, y : T_\times \vdash e : \tau \quad \Gamma_2, x : \tau, y : T_\bullet \vdash e' : \rho}{\Gamma_1\Gamma_2, y : T_\bullet \vdash \texttt{let!}\ (y)\ x :: \tau = e\ \texttt{in}\ e' : \rho}$$

Shareable values

- Unsafe again

  ```
  let buf = make ()
   in let! (buf) buf' = buf
       in let _ = free buf
           in length buf'
  ```

$$\frac{\Gamma_1, y : T_\times \vdash e : \tau \quad \Gamma_2, x : \tau, y : T_\bullet \vdash e' : \rho}{\Gamma_1\Gamma_2, y : T_\bullet \vdash \texttt{let!}\ (y)\ x :: \tau = e\ \texttt{in}\ e' : \rho}$$

Shareable values

- Unsafe again
  - ```
    let buf = make ()
      in let! (buf) buf' = buf
        in let _ = free buf
          in length buf'
    ```

Shareable reference leaks!

$$\frac{\Gamma_1, y : T_\times \vdash e : \tau \quad \Gamma_2, x : \tau, y : T_\bullet \vdash e' : \rho}{\Gamma_1\Gamma_2, y : T_\bullet \vdash \texttt{let!}\ (y)\ x :: \tau = e\ \texttt{in}\ e' : \rho}$$
    ```

Shareable values

- Unsafe again
  - ```
    let buf = make ()
     in let! (buf) buf' = buf
         in let _ = free buf
             in length buf'
    ```

Shareable reference leaks!

$$\frac{\rho \text{ safe for } T \qquad \Gamma_1, y : T_\times \vdash e : \tau \quad \Gamma_2, x : \tau, y : T_\bullet \vdash e' : \rho}{\Gamma_1\Gamma_2, y : T_\bullet \vdash \texttt{let!}\ (y)\ x :: \tau = e\ \texttt{in}\ e' : \rho}$$
    ```

Control Flow

- This should be allowed, but it isn't.
  - ```
    let x = alloc ()
      in if condition
         then update(x)
         else x
    ```

$$\cancel{\dfrac{x : \tau, x : \tau, \Gamma \vdash P}{x : \tau, \Gamma \vdash P}}$$

- This is unsafe
  - ```
    let x = alloc ()
      in if condition then free(x)
                      else ()
    ```

Control Flow

- This should be allowed, but it isn't.
  - ```
    let x = alloc ()
      in if condition
           then update(x)
           else x
    ```

$$\cancel{\frac{x : \tau, x : \tau, \Gamma \vdash P}{x : \tau, \Gamma \vdash P}}$$

- This is unsafe
  - ```
    let x = alloc ()
      in if condition then free(x)
                      else ()
    ```

$$\frac{\Gamma_1 \vdash c : \mathtt{Bool}_\sharp \quad \Gamma_2 \vdash t : \tau \quad \Gamma_2 \vdash e : \tau}{\Gamma_1 \Gamma_2 \vdash \mathtt{if}\ c\ \mathtt{then}\ t\ \mathtt{else}\ e : \tau}$$

Loops

Loops

- Our one higher order conceit
  - iteration schema are (external) higher order functions

$$\frac{\Gamma \vdash e : (\texttt{Arr}\ T)_{\bullet}}{\Gamma \vdash \texttt{map}\ e : (T \to T) \to (\texttt{Arr}\ T)_{\bullet}} \qquad \frac{\Gamma \vdash e : (\texttt{Arr}\ T)_{\times}}{\Gamma \vdash \texttt{fold}\ e : (T \to \epsilon) \to \epsilon}$$

$$\frac{\Gamma \vdash i : (\tau \to \rho) \to \sigma \quad \Gamma \vdash e : \gamma}{\Gamma \vdash i\ \texttt{with}\ e : ((\tau, \gamma) \to (\rho, \gamma)) \to (\sigma, \gamma)}$$

  - **for** loops are higher order function application (plus a lambda)

$$\frac{\Gamma_1 \vdash i : (\tau \to \rho) \to \sigma \quad \Gamma_2, x : \tau \vdash s : \rho}{\Gamma_1 \Gamma_2 \vdash \texttt{for}\ x\ \texttt{in}\ i\ \texttt{do}\ s : \sigma}$$

Loops

- Multiply all array elements by 2 (destructively)
  - `let arr’ = for x in map(arr) do x * 2`
- Sum up an array of integers:
  - ```
    let sum = for (x,y) in fold(arr) with 0
                do (x + y)
    ```
- Both at the same time
  - ```
    let arr’, sum = for (x,y) in map(arr) with 0
                      do (x*2,x + y)
    ```

Loops

- Unsafe again..
  - ```
    let y = alloc ()
      in for x in map(arr)
           do let _ = free(y)
                in x
    ```

$$\frac{\Gamma_1 \vdash i : (\tau \to \rho) \to \sigma \quad \Gamma_2, x : \tau \vdash s : \rho}{\Gamma_1 \Gamma_2 \vdash \mathtt{for}\ x\ \mathtt{in}\ i\ \mathtt{do}\ s : \sigma}$$
    ```

Loops

- Unsafe again..
  -
```
let y = alloc ()
  in for x in map(arr)
       do let _ = free(y)
          in x
```

$$\frac{\Gamma_1 \vdash i : (\tau \to \rho) \to \sigma \quad \Gamma_2, x : \tau \vdash s : \rho}{\Gamma_1\Gamma_2 \vdash \texttt{for } x \texttt{ in } i \texttt{ do } s : \sigma}\ \Gamma_2 \text{ does not contain any linear types}$$

Error Handling

Error Handling

- C has error-handling via a return-code convention
  - We can do better!

Error Handling

- C has error-handling via a return-code convention
  - We can do better!
- Solution
  - Add a separate syntactic layer, *statements*, above the expression layer.
    - Move `let!`, `for`, `let`, and `if` on to the statement level.
      - (and anonymous products)
  - Statements are different from expressions in that they can **evaluate to multiple values** and they can **fail**.

Error Handling

- C has error-handling via a return-code convention
  - We can do better!
- Solution
  - Add a separate syntactic layer, *statements*, above the expression layer.
    - Move `let!`, `for`, `let`, and `if` on to the statement level.
      - (and anonymous products)
  - Statements are different from expressions in that they can **evaluate to multiple values** and they can **fail**.

$$s : \overline{\tau_s}$$
$$s : \texttt{fails}\ \overline{\tau_f}$$
$$s : \overline{\tau_s}\ \texttt{fails}\ \overline{\tau_f}$$

Error Handling

Error Handling

$$\frac{\text{for each } i\text{: } \Gamma_i \vdash e_i : \tau_i}{\overline{\Gamma_i} \vdash \texttt{return}\ \overline{e_i} : \overline{\tau_i}}$$

Error Handling

$$\frac{\text{for each } i\text{: } \Gamma_i \vdash e_i : \tau_i}{\overline{\Gamma_i} \vdash \texttt{return}\ \overline{e_i} : \overline{\tau_i}}$$

$$\frac{\Gamma_c : e_c : \texttt{err}_\sharp \quad \text{for each } i\text{: } \Gamma_i \vdash e_i : \tau_i}{\Gamma_c \overline{\Gamma_i} \vdash \texttt{fail}\ e_c\ \overline{e_i} : \texttt{fails}\ \overline{\tau_i}}$$

Error Handling

$$\frac{\text{for each } i\text{: } \Gamma_i \vdash e_i : \tau_i}{\overline{\Gamma_i} \vdash \texttt{return}\ \overline{e_i} : \overline{\tau_i}}$$

$$\frac{\Gamma_c : e_c : \texttt{err}_\sharp \quad \text{for each } i\text{: } \Gamma_i \vdash e_i : \tau_i}{\Gamma_c \overline{\Gamma_i} \vdash \texttt{fail}\ e_c\ \overline{e_i} : \texttt{fails}\ \overline{\tau_i}}$$

$$\frac{\Gamma_1 : e : \texttt{Bool}_\sharp \quad \Gamma_2 : s_t : T_t \quad \Gamma_2 : s_e : T_e}{\Gamma_1 \Gamma_2 \vdash \texttt{if}\ e\ \texttt{then}\ s_t\ \texttt{else}\ s_e : T_t \sqcup T_e}$$

Error Handling

$$\frac{\text{for each } i\text{: } \Gamma_i \vdash e_i : \tau_i}{\overline{\Gamma_i} \vdash \texttt{return}\ \overline{e_i} : \overline{\tau_i}}$$

$$\frac{\Gamma_c : e_c : \texttt{err}_\sharp \qquad \text{for each } i\text{: } \Gamma_i \vdash e_i : \tau_i}{\Gamma_c \overline{\Gamma_i} \vdash \texttt{fail}\ e_c\ \overline{e_i} : \texttt{fails}\ \overline{\tau_i}}$$

$$\frac{\Gamma_1 : e : \texttt{Bool}_\sharp \qquad \Gamma_2 : s_t : T_t \qquad \Gamma_2 : s_e : T_e}{\Gamma_1 \Gamma_2 \vdash \texttt{if}\ e\ \texttt{then}\ s_t\ \texttt{else}\ s_e : T_t \sqcup T_e}$$

Subtyping!

$$\begin{tikzcd} & \overline{\tau_t}\ \texttt{fails}\ \overline{\tau_u} & \\ \texttt{fails}\ \overline{\tau_u} \arrow[ur] & & \overline{\tau_t} \arrow[ul] \end{tikzcd}$$

Error Handling

- Let (and Let!) only deal with the success case!

```
let x = fail(EINVAL, 3) ???
```

  - We add binding (and let!) forms for failure cases too.
  - The most interesting form is for the possible-failure case, which is also a branching construct:

$$\mathtt{handle}\ s\ (\overline{x}.\ s_s)\ (c\ \overline{x}.\ s_f)$$

Error Handling

- Let (and Let!) only deal with the success case!

```
let x = fail(EINVAL, 3) ???
```

  - We add binding (and let!) forms for failure cases too.
  - The most interesting form is for the possible-failure case, which is also a branching construct:

$$\texttt{handle}\ s\ (\overline{x}.\ s_s)\ (c\ \overline{x}.\ s_f)$$

We force you to handle your error cases!

Datatypes

Datatypes

- CDSL supports product (record) and sum (tagged union) types
  - `.{ field1 : .T, field2 : .U}`
  - `.< tag1 : T, tag2 : U >`

Datatypes

- CDSL supports product (record) and sum (tagged union) types
  - `.{ field1 : .T, field2 : .U}`
  - `.< tag1 : T, tag2 : U >`
- Product types are complicated:

Datatypes

- CDSL supports product (record) and sum (tagged union) types
  - `.{ field1 : .T, field2 : .U}`
  - `.< tag1 : T, tag2 : U >`

- Product types are complicated:

```
let sum = operation(x.field1, x.field2)
```

Datatypes

- CDSL supports product (record) and sum (tagged union) types
  - `.{ field1 : .T, field2 : .U}`
  - `.< tag1 : T, tag2 : U >`

- Product types are complicated:

```
let sum = operation(x.field1, x.field2)
```

✗

Record Types

- Need to smash open a record into its constituent fields

```
let token { f1, f2 } = open rec
    f1', f2' = update(f1, f2)
 in close token {f1 = f1', f2 = f2' }
```

Record Types

- Need to smash open a record into its constituent fields

```
let token { f1, f2 } = open rec
    f1’, f2’ = update(f1, f2)
 in close token {f1 = f1’, f2 = f2’ }
```

for destructive update

Generating Purely Functional Specs

```
SimpleObj = { a : #U8 , b : #U8, c : .Foo }

simpleobj_example (so : .SimpleObj) : .SimpleObj fails .SimpleObj
= { buf      <- buf_create(42)
                handle code { fail (code, so) }
  ; buf,i    <- let! (so) simpleobj_serialise(buf,so,0)
                handle (code, buf) { free(buf); fail (code, so) }
  ; so2      <- simpleobj_new('_',0)
                handle code { free(buf); fail (code, so) }
  ; so2      <- let! (buf) simpleobj_unserialise(buf, so2, 0)
                handle (code, so2) { free(buf,so2); fail (code, so) }
  ; ok <- let!(so, so2) return (so.a == so2.a && so.b == so2.b)
  ; free(buf)
  ; if not(ok) then { free (so2); fail (32,so) }
                else { free (so); return (so2) }
  }
```

Generating Purely Functional Specs

```
SimpleObj = { a : U8 , b : U8, c : Foo }

simpleobj_example (so : SimpleObj) : SimpleObj fails SimpleObj
= { buf     <- buf_create(42)
               handle code { fail (code, so) }
  ; buf,i   <- let! (so) simpleobj_serialise(buf,so,0)
               handle (code, buf) { free(buf); fail (code, so) }
  ; so2     <- simpleobj_new('_',0)
               handle code { free(buf); fail (code, so) }
  ; so2     <- let! (buf) simpleobj_unserialise(buf, so2, 0)
               handle (code, so2) { free(buf,so2); fail (code, so) }
  ; ok <- let!(so, so2) return (so.a == so2.a && so.b == so2.b)
  ; free(buf)
  ; if not(ok) then { free (so2); fail (32,so) }
               else { free (so); return (so2) }
  }
```

Generating Purely Functional Specs

```
SimpleObj = { a : U8 , b : U8, c : Foo }

simpleobj_example (so : SimpleObj) : SimpleObj fails SimpleObj
= { buf     <- buf_create(42)
               handle code { fail (code, so) }
  ; buf,i   <- simpleobj_serialise(buf,so,0)
               handle (code, buf) { free(buf); fail (code, so) }
  ; so2     <- simpleobj_new('_',0)
               handle code { free(buf); fail (code, so) }
  ; so2     <- simpleobj_unserialise(buf, so2, 0)
               handle (code, so2) { free(buf,so2); fail (code, so) }
  ; ok <- return (so.a == so2.a && so.b == so2.b)
  ; free(buf)
  ; if not(ok) then { free (so2); fail (32,so) }
               else { free (so); return (so2) }
  }
```


Generating Purely Functional Specs

```
SimpleObj = { a : U8 , b : U8, c : Foo }

simpleobj_example (so : SimpleObj) : SimpleObj fails SimpleObj
= { buf     <- buf_create(42)
               handle code { fail (code, so) }
  ; buf,i   <- simpleobj_serialise(buf,so,0)
               handle (code, buf) { fail (code, so) }
  ; so2     <- simpleobj_new('_',0)
               handle code { fail (code, so) }
  ; so2     <- simpleobj_unserialise(buf, so2, 0)
               handle (code, so2) { fail (code, so) }
  ; ok <- return (so.a == so2.a && so.b == so2.b)

  ; if not(ok) then { fail (32,so) }
               else { return (so2) }
  }
```

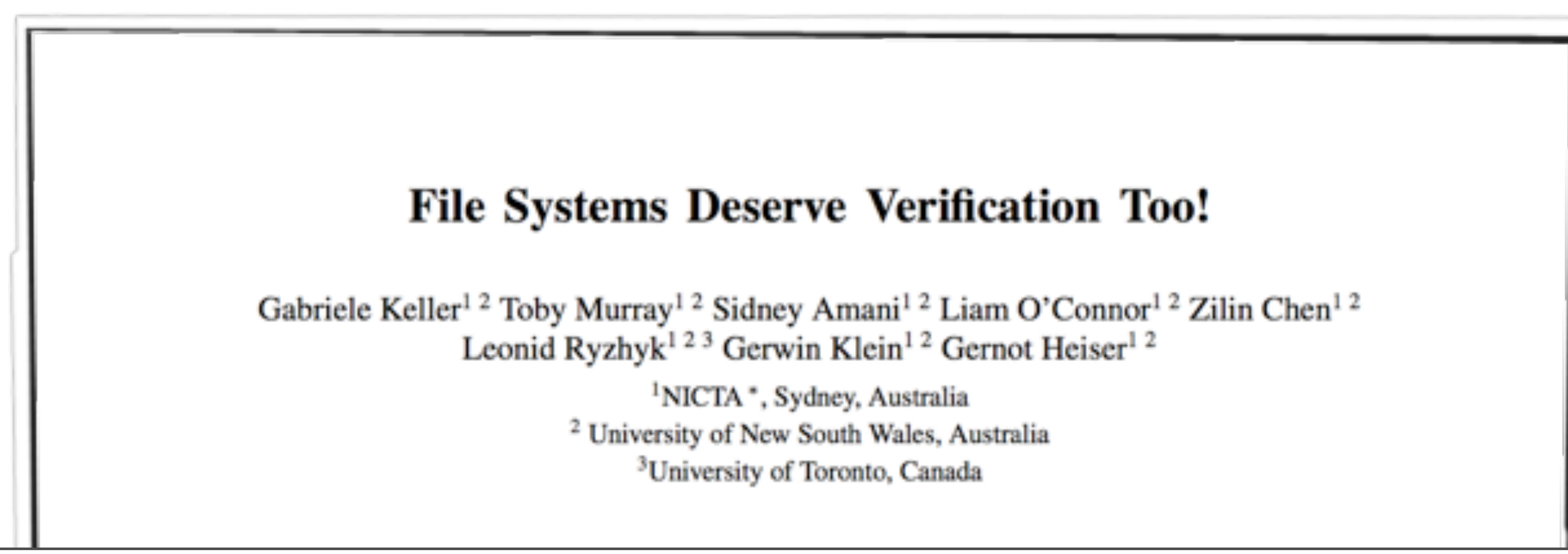
Generating Purely Functional Specs

```
SimpleObj = { a : U8 , b : U8, c : Foo }

simpleobj_example (so : SimpleObj) : (Err, SimpleObj) + SimpleObj
= case buf_create(42) of
    Inl code -> Inl (code,so)
    Inr buf  -> case simpleobj_serialise(buf,so,0) of
      Inl (code,buf) -> Inl (code,so)
      Inr (buf,i)    -> case simpleobj_new('_',0) of
        Inl code -> Inl (code,so)
        Inr so2 -> case simpleobj_unserialise(buf, so2, 0) of
          Inl (code,so2) -> Inl (code,so)
          Inr so2 -> let ok = (so.a == so2.a && so.b == so2.b)
                     in if not(ok) then Inl (32,so)
                                   else Inr (so2)
```

Current Status

- We have
  - A paper about our overall project (not just CDSL) in PLOS this year.
  - A working (but unverified) compiler to C
  - Formalised type system + dynamics on paper
  - Formalised dynamic semantics in Isabelle
  - Some outdated safety proofs in Agda
  - A good feeling about proof work remaining to be done
  - A prototype of another DSL for disk (de-)serialisation that generates CDSL
  - A syntax headache



File Systems Deserve Verification Too!

Gabriele Keller[1 2] Toby Murray[1 2] Sidney Amani[1 2] Liam O'Connor[1 2] Zilin Chen[1 2]
Leonid Ryzhyk[1 2 3] Gerwin Klein[1 2] Gernot Heiser[1 2]

[1]NICTA *, Sydney, Australia
[2] University of New South Wales, Australia
[3]University of Toronto, Canada