Systems@ETH Zürich

Domain-Specific Languages

Advanced Operating Systems
(263-3800-00)

Timothy Roscoe

Thursday 9th December 2010

Outline

Systems@ETH Zürich

- Introduction
- Interface definition languages
- Hardware interface languages
- Filet-of-Fish
- Hamlet
- References

The problem

- C is a pain to write OS code in.
- 2 classes of problem:
  1. Lack of automatic resource mgmt
  2. Hard to express high-level semantics

High-level languages to the rescue?

- Write your OS in Java/Eiffel/C#/etc.
  - Has been tried. Several times.
- Problems:
  - Lose all control over resource management
  - Explicit layout / memory access becomes hard
  - Still can’t express high-level semantics
    - (OS code is highly specialized)
  - Sufficiently-expressive languages too slow and too abstract
    - (e.g. Haskell)

Extend C?

Promising approach:

- NesC: TinyOS's C dialect with support for modules, events [Gay 2003]
- Deputy: extensions to C using type inference for static checks [e.g. Anderson 2009]
- Ivy: evolving C as a language [Brewer 2005]

So far, little uptake (poor toolchain support?)

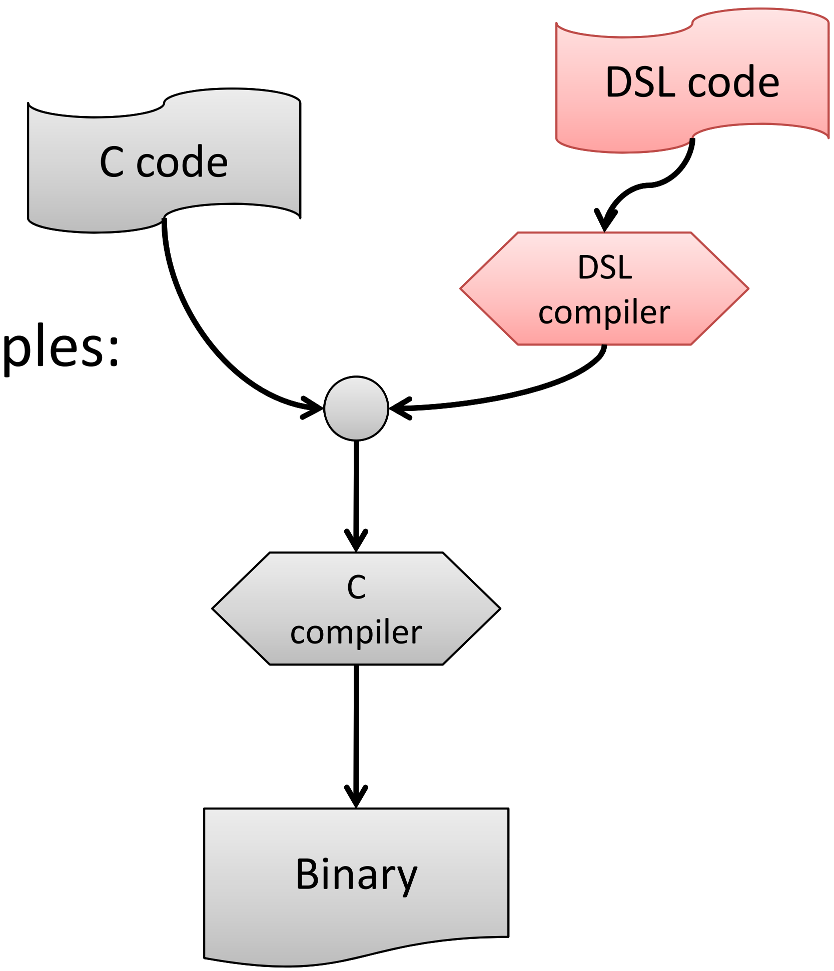
Domain specific languages

- Old idea
- Very broad applicability (not just OSes)
- Guy Steele: "design your system as if you were designing a language anyway".
- Build a "little language" tailored for the task at hand
- Generate C which is then compiled with the OS

In Barrelfish, we use DSLs extensively (4 so far, and counting)

Domain specific language workflow

For all our examples:



Advantages

- Highly specialized: capture the exact semantics you want!
- Can check and enforce useful invariants
- Small, easy to learn
- Can be very fast (faster than a programmer could write)
- Dramatically reduces devel/debug time

Of course, there is a downside:

- Lot of effort to write the compiler
- Complicates toolchain management
- May make the code look somewhat alien...

Examples of DSLs in Operating Systems

Systems@ETH zürich

- Communication interface definition
- Hardware register access
- Scheduling algorithms (see next week!)
- Protocol stack design (Click, Prolac)
- Capability type system specification
- Error code definitions
- ...

Outline

- Introduction
- Interface definition languages
- Hardware interface languages
- Filet-of-Fish
- Hamlet
- References

Interface definition languages

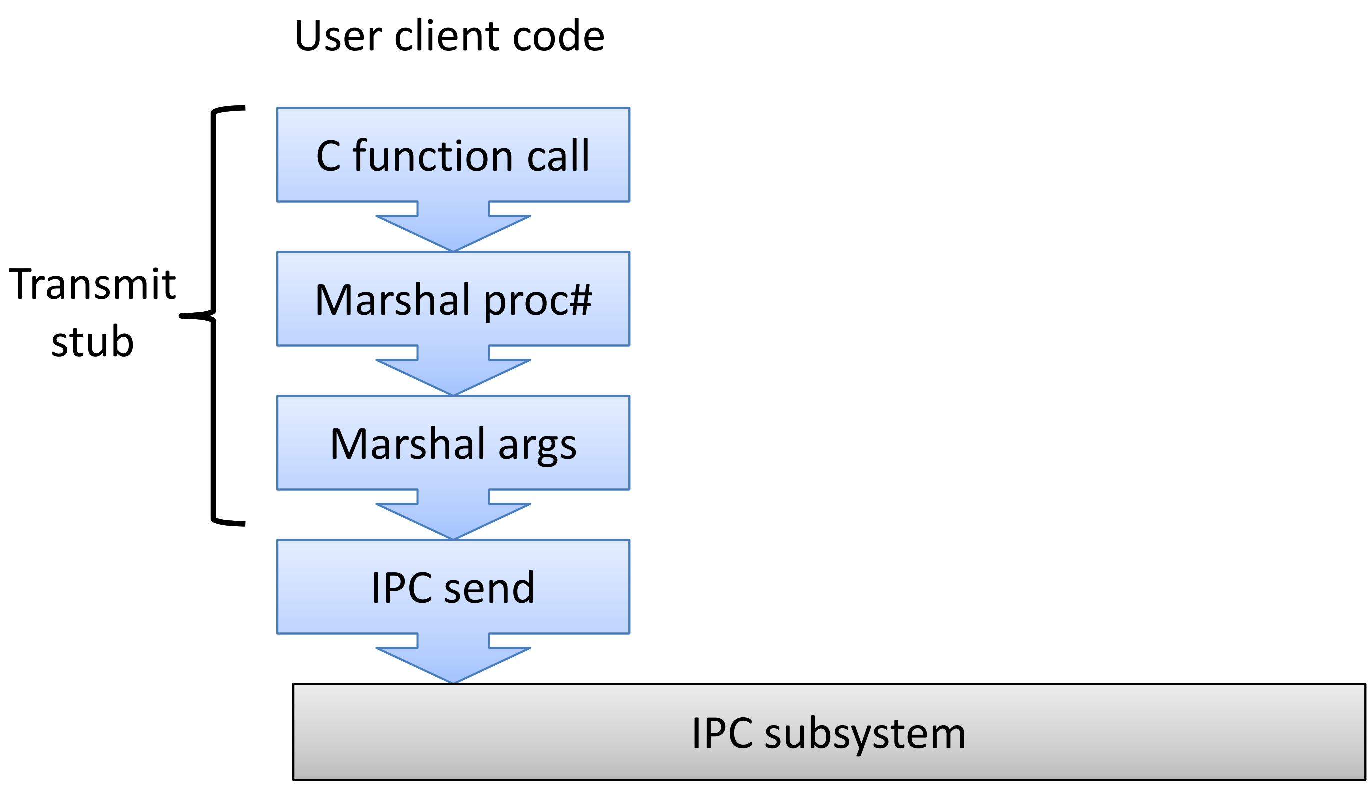
- Perhaps oldest DSLs for OS development
- Original RPC [Birrell and Nelson, 1984]

Interface definition semantics

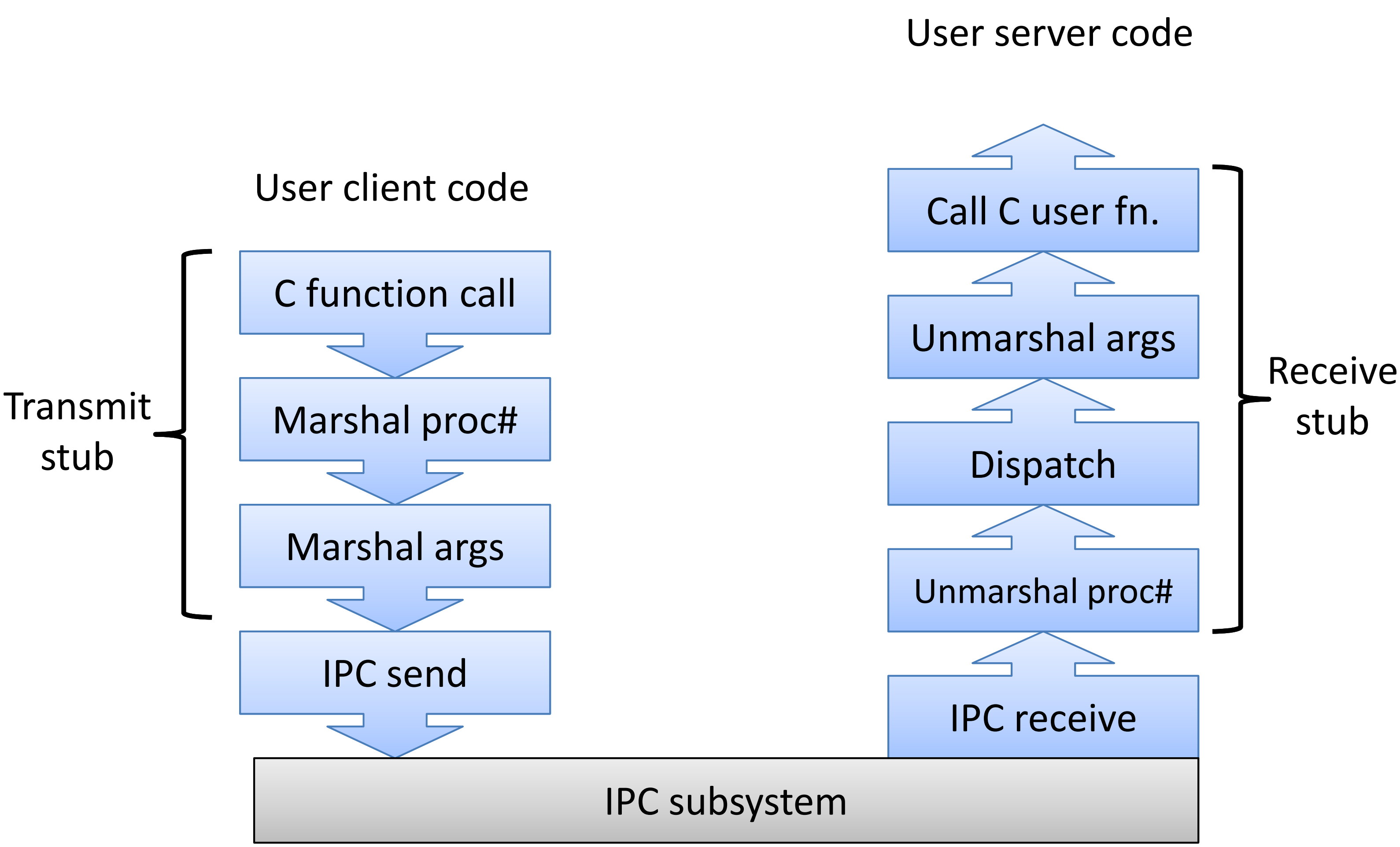
IDLs are NOT like (Java) RMI!

- An IDL typically defines its own type system.
- Concrete types: integers, structs, etc.
- Abstract types: interface references
- IDL compiler maps this to (perhaps many) programming languages

Stub functionality



Stub functionality



Memory management in IDLs

Network RPC IDLs need to worry about memory management:

- E.g. CORBA defines 3 parameter modes:
  - **in**: Argument is passed from client to server (parameter)
  - **out**: Argument is passed from server to client (result)
  - **inout**: Argument is sent to server, modified, sent back

Memory management in IDLs

Network RPC IDLs need to worry about memory management:

- E.g. CORBA defines 3 parameter modes:

  in: Argument is passed from client to server (parameter)

  out: Argument is passed from server to client (result)

  inout: Argument is sent to server, modified, sent back

- This is not enough locally, in an OS with:
  - Shared memory transport
  - No garbage collection
  - Values (like arrays) bigger than a register or machine word

Memory management in IDLs

Basic questions:

- When should memory in the client be freed?
- How can memory in the server be allocated?
- How can memory in the client be allocated?
- When should memory in the server be freed?
- When is it safe to modify client data, if it’s been sent to the server?

Memory management in IDLs

[Hamilton and Kougiouris, 1994]

The IDL for the Spring OS modified CORBA IDL for an OS setting:

- **copy**: Argument is copied to the server.
- **consume**: Argument is sent from client to server, and destroyed at client.
- **produce**: Argument is generated at the server and sent back (destroyed at server)
- **borrow**: Like **inout**, but can't be modified by client in the meantime.

Performance

For network IDLs (CORBA, ANSA, DCE, SunRPC, etc.) stub performance not critical

- Network latency dominates
- Calls are infrequent
- Calls must traverse network stack anyway

Performance

For network IDLs (CORBA, ANSA, DCE, SunRPC, etc.) stub performance not critical

- Network latency dominates
- Calls are infrequent
- Calls must traverse network stack anyway

It's very different for local OS stubs:

- IPC system highly optimized ⇒ stub performance critical
- Calls are frequent (particularly in a microkernel)

Flick

[Eide et al., 1997]

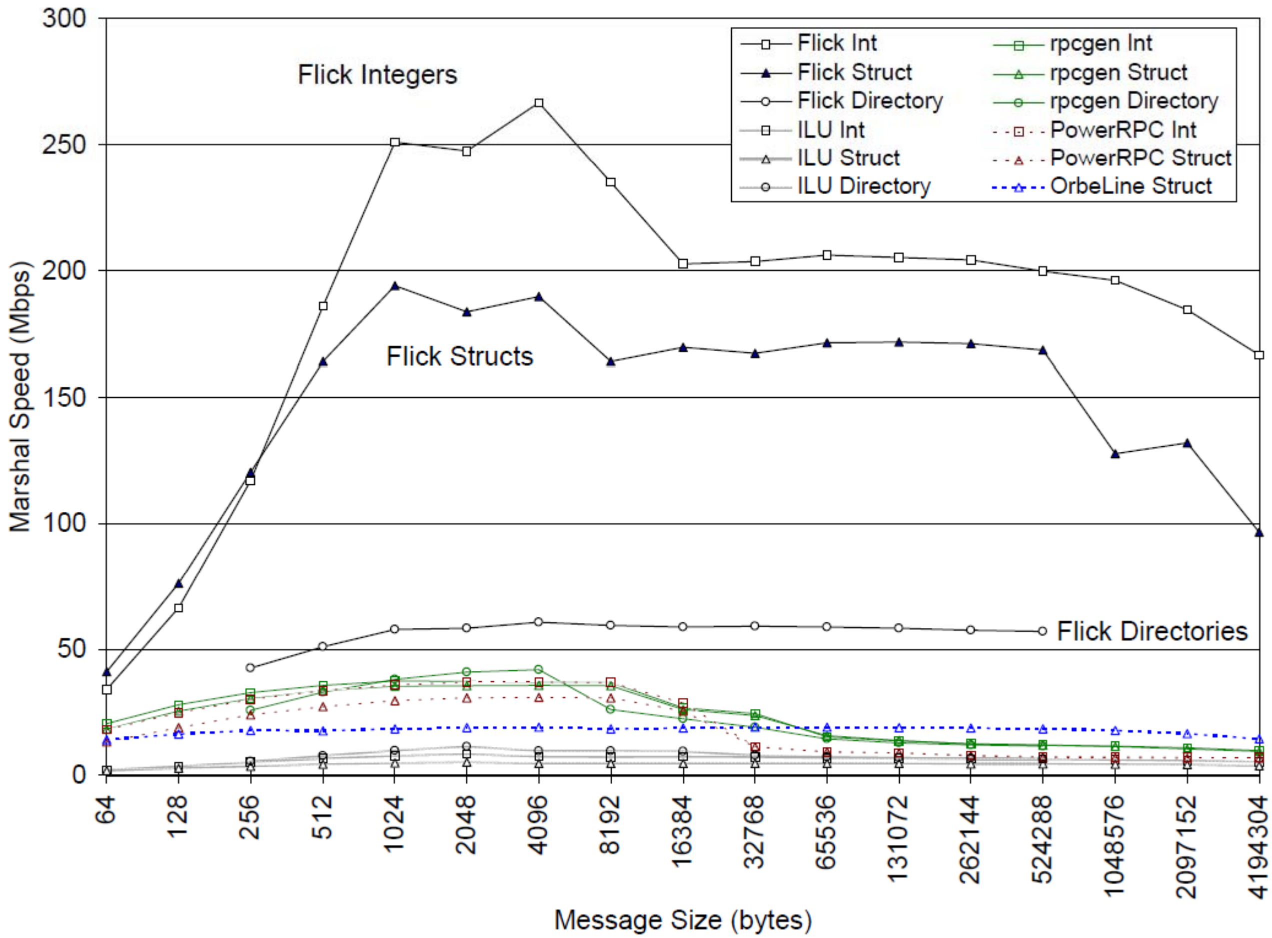
Optimizing stub compiler: many techniques, e.g.:

- Marshall all fixed-size data in one go
  - Avoid checking buffer size each time
- Inline most operations
- Use receive buffer space for arguments (e.g. in)
- Aggregate copies into one big **`memcpy`**
- Transport-specific marshalling
  - e.g. L4 IPC in registers

Flick performance

[Eide et al., 1997]

- Shows effect of compiler optimizations on marshalling code:



General pattern

Consider a domain specific language where:

- You’re writing the same boilerplate code again and again (with minor variations)
- It’s easy to make mistakes
- Interoperability (common specifications) are useful
- It’s clear what the compiler should do
- Compiler optimizations would be useful

... or at least some of the above.

Outline

- Introduction
- Interface definition languages
- Hardware interface languages
- Filet-of-Fish
- Hamlet
- References

Hardware register access

Accessing hardware registers is generally fiddly code

- Lots of bit manipulation (registers have many fields)
- Poor C support
  - word size, sign extension, volatile semantics
  - bitfield structs are implementation specific!
- Consequences of errors are bad
  - Very hard to find bugs
  - Frequently hangs entire machine
- C code to manipulate registers is tedious to write

Devil Example: Logitech Busmouse

[Mérillon et al., 2000]

- Hand-written macros:

```
#define MSE_DATA_PORT             0x23c
#define MSE_CONTROL_PORT          0x23e
...
#define MSE_READ_Y_LOW            0xc0
#define MSE_READ_Y_HIGH           0xe0
```

Devil Example: Logitech Busmouse

[Mérillon et al., 2000]

- Hand-written macros:

```
#define MSE_DATA_PORT           0x23c
#define MSE_CONTROL_PORT        0x23e
...
#define MSE_READ_Y_LOW          0xc0
#define MSE_READ_Y_HIGH         0xe0
```

- Programmer usage idioms:

```
dy = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT);
buttons = inb(MSE_DATA_PORT);
dy |= (buttons & 0xf) << 4;
buttons = ((buttons >> 5) & 0x07);
```

Devil Example: Logitech Busmouse

[Mérillon et al., 2000]

- Device specified in the Devil DSL:

```
device logitech_busmouse (base : bit[8] port @ {0..3})
{
   // Signature register (SR)
   register sig_reg = base @ 1 : bit[8];
   variable signature = sig_reg, volatile, write trigger : int(8);

   // Configuration register (CR)
   register cr = write base @ 3, mask '1001000.' : bit[8];
   variable config = cr[0] : { CONFIGURATION => '1', DEFAULT_MODE => '0' };

   // Interrupt register
   register interrupt_reg = write base @ 2, mask '000.0000' : bit[8];
   variable interrupt = interrupt_reg[4] : { ENABLE => '0', DISABLE => '1' };

   // Index register
   register index_reg = write base @ 2, mask '1..00000' : bit[8];
   private variable index = index_reg[6..5] : int(2);

   register x_low  = read base @ 0, pre {index = 0}, mask '****....' : bit[8];
   register x_high = read base @ 0, pre {index = 1}, mask '****....' : bit[8];
   register y_low  = read base @ 0, pre {index = 2}, mask '****....' : bit[8];
   register y_high = read base @ 0, pre {index = 3}, mask '...*....' : bit[8];

   structure mouse_state = {
      variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);
      variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);
      variable buttons = y_high[7..5], volatile : int(3);
   };
}
```

Devil Example: Logitech Busmouse

[Mérillon et al., 2000]

- What's Devil generating?

```
#define bm_get_mouse_state() ( \
  outb(1, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_x_high = inb(bm_cache.__dil_base__); \
  outb(0, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_x_low  = inb(bm_cache.__dil_base__); \
  outb(3, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_y_high = inb(bm_cache.__dil_base__); \
  outb(2, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_y_low  = inb(bm_cache.__dil_base__) )

#define bm_get_dy() ( \
  (bm_cache.cache_mouse_state.cache_get_y_high & 0xfu) << 4 | bm_cache.cache_mouse_state.cache_get_y_low & 0xfu)

#define bm_get_buttons() ((bm_cache.cache_mouse_state.cache_get_y_high & 0xe0u) >> 5)
```

Devil Example: Logitech Busmouse

[Mérillon et al., 2000]

- What the programmer gets to write:

```
bm_get_mouse_state();
dy = bm_get_dy();
buttons = bm_get_buttons();
```

Other Devil features

- Pre- and post-conditions
  - E.g. index registers used to access other register banks
  - Semaphores which must be held before writing a register
- "Variables"
  - values which combinations (usually concatenations) of register values

Mackerel

Barrelfish's answer to Devil

- Things have changes somewhat in the meantime:
- Lots of address space ⇒ Index registers are less frequent

  ⇒ pre-conditions less important

  Register address spaces more useful (PCI, memory, IO)

- Registers are wider (32 or 64 bits)

  ⇒ meaningful values rarely split across hardware fields

- Most complex devices communicate using descriptor rings

  ⇒ In-memory data structures are just as important as registers

Mackerel features

- Goal: specifications should be as close to datasheet descriptions as possible.

- Basic constructs specify:
  - Individual registers
  - Register types
  - Register arrays
  - In-memory data types
  - Collections of constant values

- Make extensive of C compiler's type system and inlining
- Comments are incorporated in C `printf`-like code

Mackerel features

Mackerel generates:

- C constant definitions for all constant values
- C Type definitions for all register and data types
- Functions to read/write all registers
- Functions to read/write all register and data type fields
- Functions to **`snprintf`**:
  - Register values
  - Data type values
  - Entire device state!

Mackerel

- Example: Intel e1000 Ethernet controller
- Fragment showing a register definition:

```
register status rw addr(base, 0x0008) "Device status" {
     fd       1 "Link full duplex configuration";
     lu       1 "Link up";
     lan_id   2 "LAN ID";
     txoff    1 "Transmission paused";
     tbimode  1 "TBI mode";
     speed    2 type(linkspeed) "Link speed setting";
     asdv     2 type(linkspeed) "Auto speed detection val";
     phyra    1 "PHY reset asserted";
     _        8 mbz;
     gio_mes  1 "GIO master enable status";
     _        12;
};
```

Mackerel: some figures

Lines of code (using David Wheeler's SLOCCount):

- 2359 lines of Haskell for the Mackerel compiler
- 1028 lines of Mackerel for the e1000 specification
- 23762 lines of C generated from **`e1000.dev`**

If DSLs are so good...

How come we don't see more of them in OS research?

- Quite hard to design a good one
  - Except Mackerel, all the DSLs in Barrelfish were designed ***after*** we had an initial C implementation and understood the functionality.

- Perception: the effort to implement DSL usually outweighs the cost of designing, building, and implementing it
  - With yesterday's tools, there is some truth in this
  - But . . .

Building a DSL: what does it take?

DSLs are basically simple compilers:

Building a DSL: what does it take?

DSLs are basically simple compilers:

1. Parser
   - Used to be tedious to write
   - Glorious easy these days
   - E.g. combinator-based Monadic parsing in Haskell

Building a DSL: what does it take?

DSLs are basically simple compilers:

1. Parser
   – Used to be tedious to write
   – Glorious easy these days
   – E.g. combinator-based Monadic parsing in Haskell
2. Back-end C code generator
   – Rather more difficult . . .

Writing a backend for a DSL

The backend takes an AST and generates C code

- Basically: concatenate a set of strings into a C file
- Better: encode subset of C syntax into functional combinators easier.

But still:

- Writing code through a level of indirection
- Only captures syntax of C, not intended semantics.
- Can't automate tests
- Error-prone
- Annoying to debug
- Ultimately, no assurance it works.

Filet-o-Fish

[Dagand et al., 2009]

Filet-o-Fish is . . .

- Tool for writing C code generators
- Embedding of a subset of C in Haskell
- Notation for expressing DSL semantics
- Library for creating provably-correct C code from semantic specifications

Used in Barrelfish for (to date) 2 DSLs:

Fugu defines error codes and an error stack

Hamlet defines capability type system

Hamlet: specifying the capability type system

Yes, Hamlet really is a type of fish

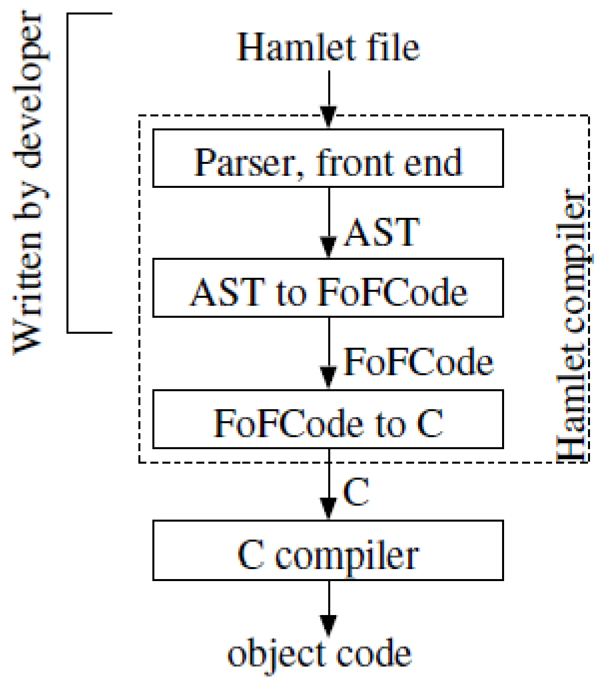
Recall that Barrelfish uses typed, partitioned capabilities

- For each capability, we must specify:
  - Physical layout in memory
  - What it can be retyped to and from
  - Valid invocations on the capability
  - What happens when it is passed between domains
  - etc.

- We capture all this information in a Hamlet specification.

How Filet-o-Fish compiles Hamlet

[Dagand et al., 2009]



How Filet-o-Fish compiles Hamlet

[Dagand et al., 2009]

- Defining semantics instead of syntax:

$$
\begin{tikzcd}
DSL \arrow[d, "\text{DSL compiler}"] & \\
FoFcode \arrow[r, "\text{FoF interpreter}"] \arrow[d, "\text{FoF compiler}"] & . \arrow[d, leftrightarrow, "\equiv"] \\
C \arrow[r, "\text{C semantics}"] & .
\end{tikzcd}
$$

What does FoF look like?

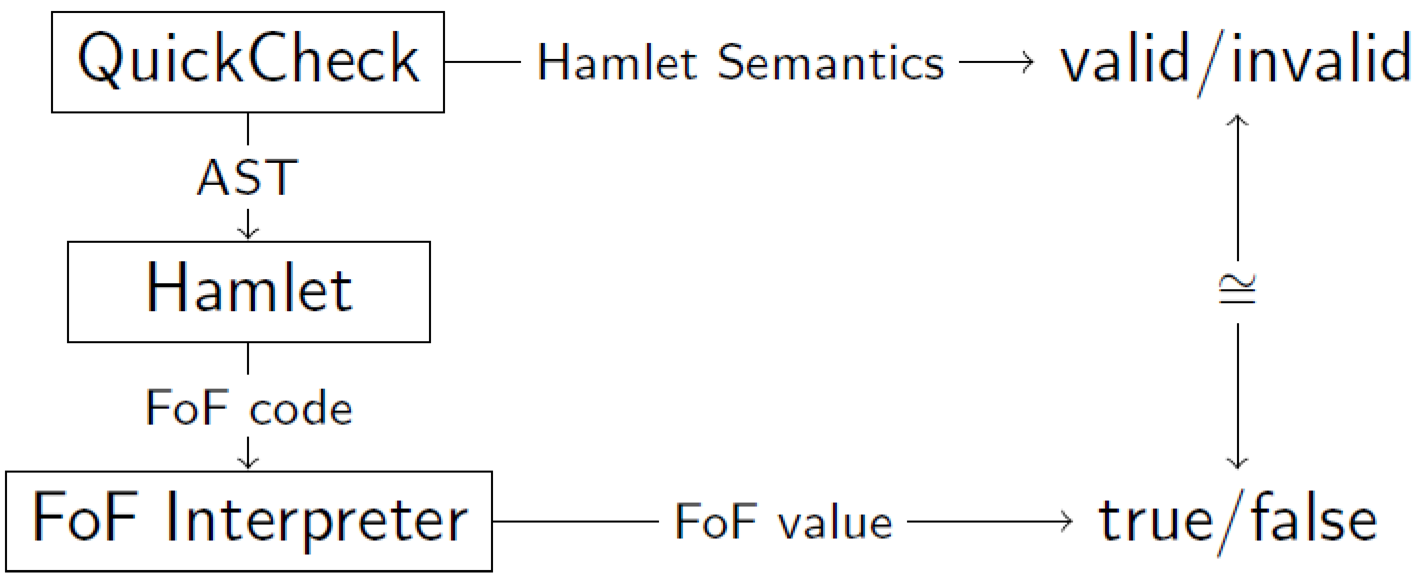
- For the previous example, Haskell resembles:

```
validateRetypeCode destType (srcType, validTypes) =
   do srcTypeV <- srcType
      validTypesP <- sequenceSem validTypes
      return (srcTypeV,
              (do returnc $ condition validTypesP))
   where condition validTypes = fold orType false validTypes
         orType acc srcType = acc .||. (destType .==. srcType)
```

Using QuickCheck to test DSLs

[Dagand et al., 2009]

- Check randomly-generated ASTs against semantic assertions:



Affecting the OS design

Hamlet makes it easy to add new capability types to Barrelfish

- Led us to encode more functionality into the type system
  - E.g. different cap types for page table levels
    (On all architectures)
  - Type system enforces page table correctness

- Can encode multiple physical address spaces, etc.
- We expect to push further functionality into capability system…

Summary

Used appropriately:

- Reduce code complexity
  – Though rarely, if never, actually evaluated
  – DSLs perhaps seen more as a means to an end...
- Reduce bugs
  – Capture (and check) high-level semantics of the domain
- Facilitate automated testing and/or correctness proofs

References

Systems@ETH Zürich

- Birrell, A. D. and Nelson, B. J. (1984). **Implementing remote procedure calls.** ACM Trans. Comput. Syst., 2(1):39–59.
- Dagand, P.-E., Baumann, A., and Roscoe, T. (2009). **Filet-o-Fish: Practical and Dependable Domain-Specific Languages for OS Development**. In Proc. 5th Workshop on Programming Languages and Operating Systems (PLOS 2009).
- Eide, E., Frei, K., Ford, B., Lepreau, J., and Lindstrom, G. (1997). **Flick: A flexible, optimizing IDL compiler**. In PLDI, pages 44–56.
- Hamilton, G. and Kougiouris, P. (1994). **The Spring nucleus: A microkernel for objects**. Technical report, Sun Microsystems Laboratories.
- Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. (2000). **Devil: An IDL for hardware programming**. In Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation.

References

- David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. **The nesC language: A holistic approach to networked embedded systems**. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03). ACM, New York, NY, USA, 1-11
- Zachary R. Anderson, David Gay, and Mayur Naik. 2009. **Lightweight annotations for controlling sharing in concurrent data structures**. In Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)
- Eric Brewer, Jeremy Condit, Bill McCloskey, and Feng Zhou. 2005. **Thirty years is long enough: getting beyond C**. In Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10 (HOTOS'05), Vol. 10. USENIX Association, Berkeley, CA, USA, 14-14.