The Engineering Challenges of Trustworthy Computing

Greg Morrisett

Harvard University

Back to the Future

Back in the early ‘90s, I was a graduate student at CMU.

There were two *very* exciting groups:
- The PL group (SML)
- The OS group (Mach)

I hung out with people from both groups.

Microkernels

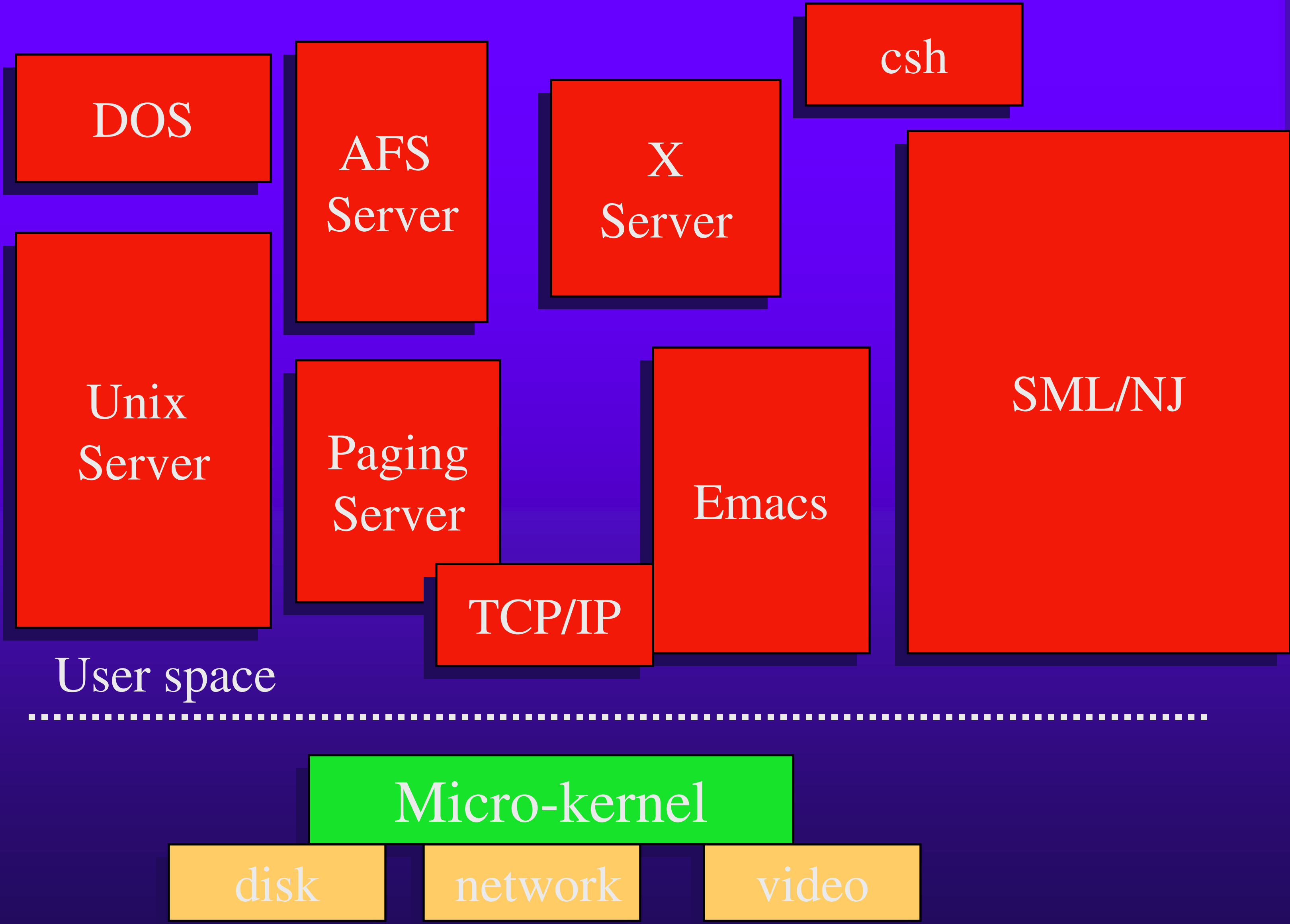
Operating systems in the 70's were small and relatively trustworthy: "kernels"

Operating systems in the 80's (e.g., Ultrix) were big, unreliable, inflexible.

- or glorified device drivers (e.g., DOS)

The Mach guys were going to fix all that…

My Mach Workstation

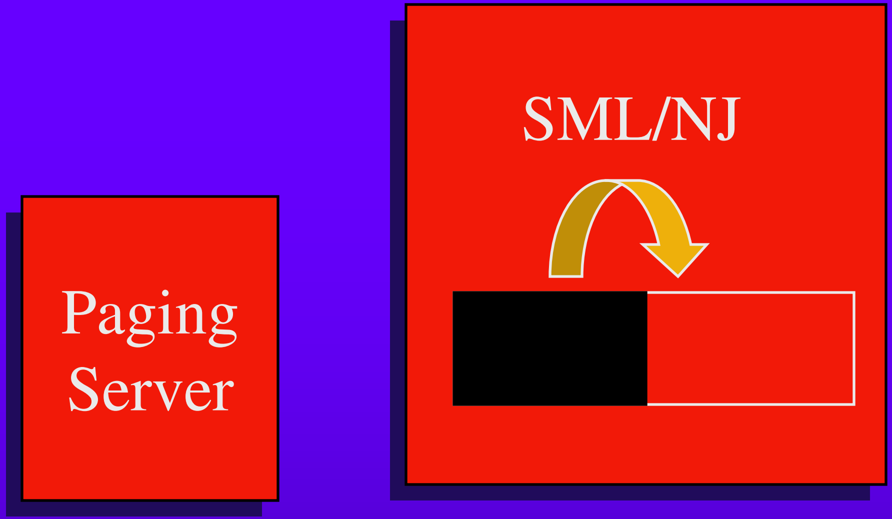


The Key Idea:

Put everything possible at the *user-level*.

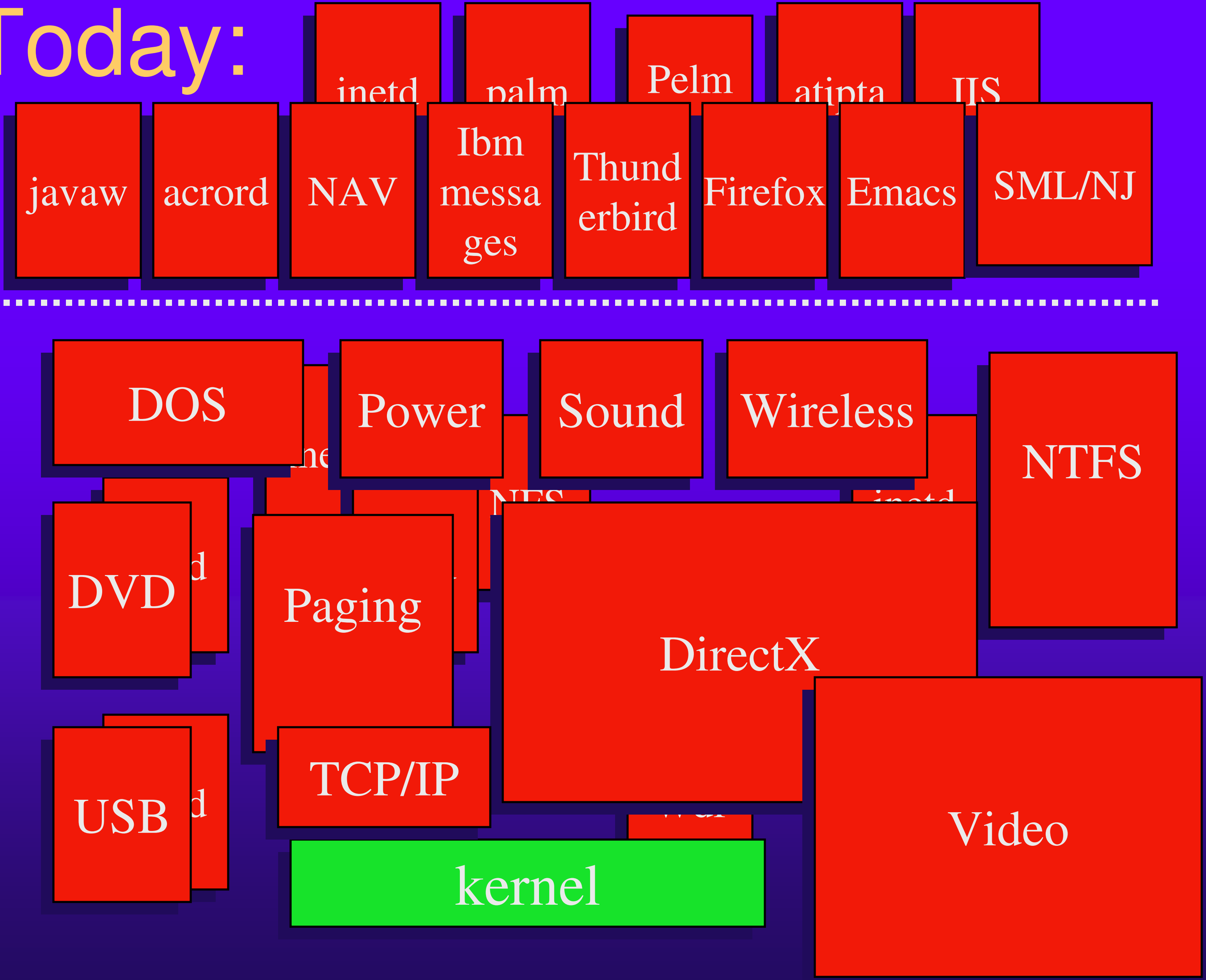
- Minimizes the *Trusted Computing Base*.
  - Less code has to be right to keep the system running and secure.
  - Isolates failures -- when DOS crashes, it shouldn't affect my UNIX server.
- Flexible
  - Easy to add/remove new services.
  - Multiple personalities.
  - Application-specific services.

Flexibility: User-Level Paging



- SML/NJ wants to do a copying collection.
- After GC, the "old" space is no longer needed.
- A conventional pager would write those pages to disk and read them back in for the next GC.
- Our custom pager just dropped the pages and provided a zero-filled page at next GC.

Today:


inetd
palm
Pelm
atipta
IIS
javaw
acrord
NAV
Ibm messa ges
Thund erbird
Firefox
Emacs
SML/NJ
DOS
Power
Sound
Wireless
NTFS
DVD
Paging
DirectX
TCP/IP
USB
kernel
Video

Today's Systems

- The OS is millions of lines of code.
  - Much of it is 3rd party (drivers, modules).
  - All written in C or assembly.
  - No isolation -- one buffer overrun ruins everyone's day.
- Most of the user-level processes and DLLs run with the *same* privileges.
  - So there's no real isolation.
  - I don't know what half of them can do.
  - Many more things are "executable"
    - ps, pdf, doc, xls, html, …

What went wrong?

- There are other ways to be flexible
  - Dynamically loadable drivers & modules.
  - Hypervisor (e.g., vmware) can give us multiple personalities.
- Performance
  - Twice as many context switches.
  - Twice as much copying.
  - It’s very easy to measure performance.
- Who cares?
  - Not so easy to measure reliability/security.
  - Back then, attacks weren’t frequent because the economic stakes weren’t that high.

SysCall in Monolithic Kernel

User Process | **Kernel**

calls `f=fopen("foo")`

library executes "break" → trap → saves context, flushes TLB, etc.

checks UID against ACL, sets up IO buffers & file context, pushes ptr to context on user's stack, etc.

restores context, clears supervisor bit →

calls `fread(f,n,&buf)`

library executes "break" → saves context, flushes TLB, etc.

checks f is a valid file context, does disk access into local buffer, copies results into user's buffer, etc.

restores context, clears supervisor bit →

Time ↓

SysCall in Mach

User Process

`f=fopen("foo")`

"break"

Kernel

saves context

checks capabilities, copies arguments switches to Unix server context

Unix Server

checks ACL, sets up buffers, etc.

"returns" to user.

Kernel

saves context

checks capabilities, copies results

restores user's context

Time

Nothing special about OSs

- Strong isolation is always a good idea from a security & reliability standpoint.
- But the costs always seem to lead us to tear down the walls:
  - Originally, web servers forked a separate process with attenuated capabilities for each CGI request.
  - The overheads of the fork were seen as too expensive, so the script is run in the context of the server…
  - Same story for web clients, databases, …

The Challenges

We need a new architecture for security.

- We need better security policies.
  - Centralized, end-to-end: my credit card should not go out the door without my consent, and never in the clear and never to an untrusted 3rd party.
  - The policies need to be phrased at the level of *applications* and *humans,* not OS objects or keys.
  - They need to be simple and easily [re]configured.
- We need better enforcement mechanisms.
  - Avoid the overheads that tempt us back into a monolithic mind set.
  - Minimize the amount of code or data that we have to trust.

These Lectures

Software-based policy enforcement:

- Dynamic Isolation (SFI)
- Static Isolation (PCC)
- Certifying Compilation (TAL)
- Legacy code (Stackguard, Ccured, Cyclone)

Other good topics we don’t have time for:

- Authorization (Stack Inspection, IRMs)
- Confidentiality (SLam, Jif)

Language-Based Security

The topics I'm covering are really a subset of language-based security (software security.)

There are many other exciting areas where languages, analysis, compilers, and semantics are informing security:

- Policy languages and logics (e.g., BAN logic)
- Models and protocols (e.g., Spi)

Software-Based Memory Isolation

Greg Morrisett
Harvard University

Some References:

R.Wahbe *et al*. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symp. on Operating Systems Principles*, pages 203--216, December 1993.

C.Small. MiSFIT: A Tool for Constructing Safe Extensible C++ Systems, In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, Oregon, June 1997.

S.McCamant & G.Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Usenix Security Symposium*, Vancouver, British Columbia, August 2006.

Motivation

Many systems need to run 3rd party code.

- Kernels: drivers, modules, …
- Servers: servlets, scripts, …
- Although “trusted”, not “trustworthy”

The *right* way to do this is to fork:

- Keeps kernel/server state *isolated.*
- Can attenuate rights (e.g., change uid.)
- Can ensure availability by using only asynchronous communication.

The Problem

Everyone uses shared memory and threads:

- A process fork costs a lot compared to a thread (usually thread pool)?
- Communication costs are greater:
  - Signals, pipes, etc: must go through kernel.
  - We end up copying data to and from the server.
  - We can use shared pages and copy-on-right *if* the data are relatively large and contiguous *and* there's no sensitive data on the pages.
  - In general, about 100x the cost of a proc. call.
- In some environments, it's not an option.
  - e.g., PDA or Phone without VM support.

The Question:

From a security perspective, strong (process-based) isolation is better than shared memory and threads.

Can we avoid the overheads and limitations of the OS/Hardware-based enforcement mechanisms and still achieve isolation?

Software Fault Isolation (SFI)

- Wahbe et al. (SOSP'93)
- Use a *software-based* reference monitor to isolate components into logical address spaces.
  - conceptually: check each read, write, & jump to make sure it is within the component's logical address space.
  - hope: communication as cheap as procedure call (minimize copying, costs of context switching, etc.)
  - worry: overheads of checking will swamp the benefits of communication.
- Note: doesn't deal with other policy issues
  - e.g., availability of CPU, attenuation of rights, etc.
  - Not ideal, but definitely an improvement over shared memory services.

One Way to SFI

```
while (true) {
     if (pc >= codesz) exit(1);
     int inst = code[pc], rd = RD(inst), rs1 = RS1(inst),
          rs2 = RS2(inst), immed = IMMED(inst);
      switch (opcode(inst)) {
        case ADD: reg[rd] = reg[rs1] + reg[rs2]; break;
        case LD:  int addr = reg[rs1] + immed;
                  if (addr >= memsz) exit(1);
                  reg[rd] = mem[addr];
                  break;
        case JMP: pc = reg[rd]; continue;

      ...
    }
    pc++;
}
```

```
0: add r1,r2,r3
1: ld r4,r3(12)
2: jmp r4
```

Pros & Cons of Interpreter

Pros:

- easy to implement (small TCB.)
- works with binaries (high-level language-independent.)
- easy to enforce other aspects of OS policy

Cons:

- terrible execution overhead (x25? x70?)

but it’s a start…

Partial Evaluation (PE)

A technique for speeding up interpreters.

- At load time, we know what the code is.
- So *specialize* the interpreter to the code.
  - unroll the loop – one copy for each instruction
  - specialize the switch to the instruction
  - compile the resulting code
  - Voila!

Example PE

Original Binary:

```
0: add r1,r2,r3
1: ld r4,r3(12)
2: jmp r4
 ...
```

Interpreter

```
while (true) {
  if (pc >= codesz) exit(1);
  int inst = code[pc];
  ...
}
```

Specialized interpreter:

```
  reg[1] = reg[2] + reg[3];
  addr = reg[3] + 12;
  if (addr >= memsz) exit(1);
  reg[4] = mem[addr];
  pc = reg[4]
```

Resulting Compiled Code

```
0: add r1,r2,r3
1: addi r5,r3,12
2: subi r6,r5,memsz
3: jab _exit
4: ld r4,r5(0)
 ...
```

Control Flow is an issue…

- Suppose the source code is:

```
0x00:  beq r3,r4,42
…
0x41:  ld r2,r1(0)
0x42:  addi r2,r2,1
```

- We'll insert code before the ld, so we have to adjust the branch target to the new address.

Computed Jumps?

- In general, we may not know where we're jumping at translation time:

```
0x00:  ld r3, r1(10)
0x01:  jmp r3
…
0x41:  ld r2,r1(0)
0x42:  addi r2,r2,1
```

- So how do we adjust the jump?

Another Way to See This

```
while (true) {
    if (pc >= codesz) exit(1);
    int inst= code[pc], rd= RD(inst), rs1= RS1(inst),
        rs2 = RS2(inst), immed = IMMED(inst);
     switch (opcode(inst)) {
       case ADD: reg[rd] = reg[rs1] + reg[rs2]; break;
       case BEQ: if (reg[rs1] == reg[rs2]) pc += immed; break;
       …
       case JMP: pc = reg[rd]; continue;

   }
   pc++; }
```

- Except for JMP, the pc's value only depends on statically known quantities.
- Need a mapping T from source instruction addresses to target instruction addresses.

Computed Jumps

- But for computed jumps, we will *not* in general be able to apply T at translation time.
- Rather, we'll have to apply T at run-time if we hope to preserve the semantics of the original code.
- So **`jmp r`** in the worst case would have to become "**`r := T(r)`**"; **`jmp r`**

Ways around this…

- Most computed jumps are actually procedure "returns".
  - The return address is computed dynamically as a pc-relative offset.
  - So no translation is necessary.
- We can restrict T to proper "entry-points" in the code (e.g., labels.)
  - We lose the “any possible” binary.
  - But this is good -- we’ve cut down the possible behaviors by restricting the possible control-flow.
- Or, we can do something very, very, very clever…

Shades of Isolation

- RWJ isolation: if the program attempts to read, write, or jump to a bad address, halt.
- WJ isolation: if the program attempts to write or jump to a bad address, halt.
  - Sacrifices *confidentiality*.
  - Many more reads than writes/jumps, so faster.
- Sandboxing (Wahbe et al.): if the program attempts to write or jump to a bad address, change the address to a good one.
  - Don’t halt -- let the program keep running.
  - Sacrifices *fail-stop* detection.

How to Sandbox Writes

- Data for a domain placed in a contiguous segment.
    - Upper bits form a *segment id.*

```
0x42XXXXXX
```

- Inserts code to *mask* effective address

```
Mem[EA]:= v
```

→

```
EA := EA & 0x42FFFFFF
Mem[EA] := v
```

- No branch penalty
- But you don't detect a bad write.

Sandboxing Jumps

- Again, mask destination address.
- But we also need to ensure that you don’t jump into the middle of a masking operation.

```
0x000: z := 0xbad
0x004: r := 0x104
0x004: r := r & 0x00FFFFFF
0x008: jmp r

…

0x100: z := z & 0x42FFFFFF
0x104: Mem[z] := v
```

The Trick

- Dedicate two registers, **`s`** and **`j`**.
- Invariants:
  - Register **`s`** always contains a valid data address, and **`j`** always contains a valid code address.
  - All writes use register **`s`** as the effective address, and all (computed) jumps use **`j`** as the effective address.
- Also need other scratch registers to compute effective addresses.

Example

Original Code:

```
z := 0xbad
jmp r
…
Mem[z] := v
```

Transformed Code:

```
z := 0xbad
j := r & 0x00FFFFFF
     jmp j
     …
     s := z & 0x42FFFFFF
     Mem[s] := v
```

Performance Results:

Protection vs. Sandboxing:

- RWJ isolation:
  - required 5 dedicated registers, 4 instruction sequence
  - 20% overhead on 1993 RISC machines
- WJ sandboxing:
  - requires only 2 registers, 2 instruction sequence
  - 5% overhead

Remote Procedure Call:

- 10x faster than a really good OS RPC

Sequoia DB benchmarks: 2-7% overhead for SFI compared to 18-40% overhead for OS.

What about the x86?

Some serious problems:

- The most common computed jump is a RET.
  - RET pops an address from the stack & jumps to it.
  - Rewriting this to an explicit pop, mask, & jump destroys performance.
- Small # of registers:
  - Can't afford to dedicate 2 (much less 5).
- Variable length instructions:
  - We could jump into the "middle" of an instruction.
  - The "middle" may parse as a different instruction.

A Solution (see McCamant paper)

- Force all computed to jumps to be to k-byte-aligned addresses (e.g., k=16).
- Perform masking/etc. *within* k-byte chunk.
- No need for strong invariants on **`s`** & **`j`**: must assume any register should be masked on entry to a chunk, if it's to be used in a store or write.
- (Bunch of other low-level hacks to minimize overheads.)

Some Thoughts

- Software-based isolation is a form of *code rewriting*.
  - Input: untrusted code
  - Output: code respecting invariants that imply our policy (memory isolation).
- But our policy is pretty weak.
  - For instance, we can’t hand out memory/code in small pieces (need a contiguous chunk.)
  - Can we push this idea for stronger policies?
  - In particular, can we control down to the bit-level what can/can’t be accessed?
  - And what about availability, confidentiality, etc.?

More Thoughts

The invariants for the x86 are quite tricky.

– So tricky, Steven constructed a machine-checked proof using ACL2 that they were in fact invariants.

And the rewriter is a mess of perl code.

– I’m not sure that I trust the code is correct.

– More sophisticated policies, will demand more sophisticated rewriters.

– Even *less* likely that they will be correct.

Coming Up:

Can we *statically* verify isolation of machine code?

Can we enforce stronger security policies than isolation?

How do we do this without getting a huge trusted computing base?

Proof-Carrying Code

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

June 15, 2007

References:

- G. Necula, Proof-Carrying Code. In Benjamin C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, MIT Press, 2005.

- G. Necula, Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, Paris, France, pp. 106–119, January 1997.

- G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *USENIX Symposium on Operating System Design and Implementation*, Seattle, Washington, pp. 229–243, Oct. 1996.

Motivation:

There were some problems with using types to check that code is appropiately isolated:

- Compilation of some language features, such as array-bounds-tests and type-tag-tests, are inherently *relational*.

- For instance, to check that `A[i]` is safe, we need to know that `0 <= i < A.size`.

- But our TAL types are inherently *propositional*.

- You can code around this (see DTAL), but it's awkward.

- Furthermore, the resulting type system is *complicated* and thus not exactly trustworthy.

Following Necula and Lee, we're going to apply the ideas behind *Hoare Logic* to try to avoid these short-comings.

Relational Semantics at the Machine Level

We can model the behavior of machine instructions as *predicate transformers*:

$$[\![\iota]\!] : \mathsf{MProp} \rightarrow \mathsf{MProp}$$

Intuitively, if we start in a machine state M_1 such that $M_1 \models P$, and run ι in M_1, then we'll end up in a state M_2 such that $M_2 \models [\![\iota]\!]P$.

We're going to instrument this model with our security policy to distinguish "good" code from "bad" code.

An Instrumented Semantics

In particular, instructions will not only describe their effect on the machine state, but also what conditions are necessary to safely run the code.

$$\mathcal{V}[\![\iota]\!] : \mathsf{MProp} \to (\mathsf{MProp} \times \mathsf{MProp})$$

For example, given an SFI-like policy, the $\mathcal{V}$ for a memory write will generate a condition that the address being written is in the domain of the extension.

We can lift this up to instruction sequences as follows:

$$\begin{aligned} \mathcal{V}[\![\iota; I]\!]\, P = \ & \text{let } (C, Q) = \mathcal{V}[\![\iota]\!]\, P \\ & \text{in } C \wedge \mathcal{V}[\![I]\!]\, Q \end{aligned}$$

Then we can use $\mathcal{V}$ to calculate a set of verification conditions C for each basic block which if true, imply that the code will respect the policy.

PCC Protocol

- The code consumer publishes a predicate P describing the acceptable pre-condition for the entry-point (c.f., `main`) of the code.

- The code producer takes the code heap H and:
  - generates C using $\mathcal{V}$ starting at `main`.
  - constructs a proof derivation $\mathcal{D}$ of C.
  - Ships $\langle H, \mathcal{D} \rangle$ to the code consumer.

PCC Protocol continued

- The code consumer takes the pair $\langle H, \mathcal{D} \rangle$ and:
  - re-calculates C using $\mathcal{V}$ starting at `main`.
  - verifies that $\mathcal{D}$ is a proof of C.

PCC Advantages

- The pair $\langle H, \mathcal{D} \rangle$ is "tamper-proof".
  - This is because the consumer *re-runs* $\mathcal{V}$ to get C.
  - If you modify $\mathcal{D}$ then either it will be detected as an ill-formed proof or a proof of the wrong C.
  - If you modify H in a policy-violating way, then when the consumer runs $\mathcal{V}$, it will get a different C' than what $\mathcal{D}$ proves.
  - If you modify H in a semantics-preserving fashion, you may get the same C. But then who cares?

- In principle, any code that is *provably* safe can be produced.
  - No artificial limits on performance, expressibility.
- There's nothing really limiting us to isolation policies!

The Logic

We'll be using a variant of 1^{st}-order logic

types	τ	$::=$	word | mem
exprs.	e	$::=$	$\mathtt{r} \mid x \mid n \mid \mathsf{sel}(e_m, e_a) \mid$
			$\mathsf{emp} \mid \mathsf{upd}(e_m, e_a, e_v) \mid f(e_1, \ldots, e_n)$
props.	P	$::=$	$\mathsf{true} \mid \mathsf{false} \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \mid$
			$\forall x{:}\tau.P \mid \exists x{:}\tau.P \mid \mathsf{addr}(e_a) \mid e_1 =_\tau e_2 \mid e_1 < e_2 \mid$
			$p(e_1, \ldots, e_n)$

Memory

In the logic, we're going to model memory as if it's one big (functional) array built out of upd constructors.

The sel operation performs a lookup:

$$\frac{e = e'}{\mathsf{sel}(\mathsf{upd}(m, e, v), e') = v}$$

$$\frac{e \neq e'}{\mathsf{sel}(\mathsf{upd}(m, e, v), e') = \mathsf{sel}(m, e')}$$

Using functional update supports equational reasoning as well as relating "past memories" to the present.

The Policy

The addr predicate is used to assert that a particular address is safe to read and/or write.

For instance, following SFI, we might have:

$$\frac{\mathsf{base} \leq a < \mathsf{base} + \mathsf{segsize}}{\mathsf{addr}(a)}$$

VC Generation: Part I

$$\mathcal{V}[\![r_d := v; I]\!]\, P \;=\; (\text{True},\ \exists x. P[x/r_d] \wedge r_d = v[x/r_d])$$

$$\mathcal{V}[\![r_d := r_s + v]\!]\, P = (\text{True},\ \exists x. P[x/r_d] \wedge r_d = (r_s + v)[x/r_d])$$

In both cases, the policy doesn't care so there's no verification condition.

Example:

- $\mathcal{V}[\![r_1 := r_1 + 1]\!]\, (r_1 = r_2) = (\text{True}, \exists x. x = r_2 \wedge r_1 = x + 1)$

VC Generation: Part II

$$\begin{array}{l}\mathcal{V}[\![r_d := \mathsf{Mem}[r_s + i]]\!]\, P = \\ \qquad (P \Rightarrow \mathsf{addr}(r_s + i), \\ \qquad \exists x.P[x/r_d] \wedge r_d = \mathsf{sel}(\mathsf{Mem}, r_s[x/r_d] + i))\end{array}$$

$$\begin{array}{l}\mathcal{V}[\![\mathsf{Mem}[r_s + i] := r_t]\!]\, P = \\ \qquad (P \Rightarrow \mathsf{addr}(r_s + i), \\ \qquad \exists x.P[x/m] \wedge \mathsf{Mem} = \mathsf{upd}(x, r_s + i, r_t))\end{array}$$

For both reads and writes, we require that the pre-condition implies the effective address is valid.

VC Generation: jumps?

One attempt:

$$\mathcal{V}[\![\, \text{jump}\, \ell]\!]\, P = \mathcal{V}(H(\ell))$$

But then V may not terminate:

```
   L: jump L
```

In general, we need to break loops with an *invariant*.

Recall that for TAL, we had a mapping Ψ from labels to types.

VC Generation: Jumps

Here, we'll assume that we have a context Ψ mapping labels to *pre-conditions* (predicates on machine states).

So the VC for a jump to a particular label is just:

$$\mathcal{V}[\![\mathsf{jump}\,\ell]\!]\,P = P \Rightarrow \Psi(\ell)$$

Note that if ℓ is not in Ψ, then $\mathcal{V}$ is undefined (control-flow isolation).

VC Generation: Branches

Branches are similar to jumps:

$$\begin{array}{l}\mathcal{V}[\![\textsf{ifeq } r \textsf{ jump } \ell]\!]\, P = \\ \qquad ((P \wedge r = 0) \Rightarrow \Psi(\ell), P \wedge r \neq 0)\end{array}$$

except that we take the test into account.

This refinement is a key difference with TAL, as it gives us a form of *path-sensitivity*.

VC Generation: Jumps Revisited

What about jumps through a register?

We can apply "the trick":

$$\mathcal{V}[\![\mathsf{jump}\, r]\!]\, P = P \Rightarrow ((r = \ell_1 \wedge \Psi(\ell_1)) \vee \cdots \vee (r = \ell_n \wedge \Psi(\ell_n)))$$

where $\{\ell_1, \ldots, \ell_n\} = Dom(\Psi)$.

The trick is a lot like defunctionalization—turn a jump through a function pointer into a case statement that branches to some known targets.

Note that this only works for *closed* heaps and generates a *big* predicate. Not very modular...

Avoiding Big Predicates

Suppose we annotate each jump with a set of labels $X \subseteq Dom(\Psi)$:

$$\begin{aligned}&\mathcal{V}[\![\textsf{jump}\, r_{\{\ell_1,\ldots,\ell_k\}}]\!]\, A = \\ &\qquad A \Rightarrow ((r = \ell_1 \wedge \Psi(\ell_1)) \vee \cdots \vee (r = \ell_k \wedge \Psi(\ell_k)))\end{aligned}$$

Through control-flow analysis (or types!), the compiler can usually cut X down to a small set. Of course, it also helps to make sure that all of those labels in X have the same pre-condition P, for then:

$$\mathcal{V}[\![\textsf{jump}\, r_X]\!]\, A = A \Rightarrow ((r \in X \wedge P))$$

An Aside

What we *really* want for jumps through registers is something like:

$$\mathcal{V}[\![\text{jump } r]\!]\, P = P \Rightarrow existsQ.\Psi(r) = Q \wedge Q$$

But note that this demands moving to a higher-order logic.

Even so, this is not sufficient for a *modular* treatment of computed jumps, since it requires a global Ψ.

There are ways around this (see Zhong Shao and Andrew Appel's work) but they involve introducing types similar to what you see in TAL.

Necula simply assumes we have a `return` instruction which must establish a function's post-condition.

PCC Protocol (Revised)

- The code producer takes the heap H mapping labels (ℓ) to instruction sequences I *and a context* Ψ mapping labels to assertions:
  - generates $C = \wedge_{\ell \in Dom(\Psi)} \mathcal{V}[\![H(\ell)]\!]\,\Psi(\ell)$.
  - constructs a proof $\mathcal{D}$ of C.
  - Ships $\langle H, \Psi, \mathcal{D}\rangle$ to the code consumer.

- The code consumer takes the pair $\langle H, \Psi, \mathcal{D}\rangle$ and:
  - re-generates $C = \wedge_{\ell \in Dom(\Psi)} \mathcal{V}[\![H(\ell)]\!]\,\Psi(\ell)$.

  - verifies that $\mathcal{D}$ is a proof of C.

- In practice, the consumer will also check that $\Psi(\texttt{main}) = P$ where `main` is the entry point, and P describes the context in which we'll run the code.

Issues to Resolve

- How do we represent and check proofs ($\mathcal{D}$)?
- What if we want fine-grained memory isolation as in TAL?
- What axioms do we add to support proofs of verification conditions?
- How does the code producer construct the proof?

Representing and Checking Proofs

Necula and Lee used a variant of LF to encode their logic:

- assertions become (dependent) types: T.
- proofs become LF terms: e.
- proof-checking becomes LF type-checking: $\vdash_{LF} e : T$.

From a security standpoint, the advantage of LF is that we can write a small, simple, and thus trustworthy type-checker.

LF: Expressions and Formulae

```
exp : type.
const : int -> exp
plus : exp -> exp -> exp.
sel : exp -> exp -> exp.
...

form : type.
true : form.
eq : exp -> exp -> form.
and : form -> form -> form.
forall : (exp -> form) -> form.
exists : (exp -> form) -> form.
...
```

LF: Proof Rules

```
pf : form -> type.

true_i : pf true.
refl : Π E.pf (eq E E).
sym : Π E1,E2.pf (eq E1 E2) -> pf (eq E2 E1).
and_i : Π A,B.pf A -> pf B -> pf (and A B).
and_el : Π A,B.pf (and A B) -> pf A.
and_er : Π A,B.pf (and A B) -> pf B.
forall_i : Π P.(Π E.pf (P E)) -> pf (forall P).
forall_e : Π E,P.pf (forall P) -> pf (P E).
...
```

So a proof of P is an LF term built out of these constructors with type `pf` P.

Pros and Cons of LF

An advantage of LF is that it's logic independent:

- Meaning it's easy to add new axioms, inference rules.
- Or to encode other kinds of logics (e.g., H.O. logic.)
- But proof-checking remains the same—just LF type-checking.
- And the minimal nature of LF makes it relatively easy to establish adequacy of the encoding.

Of course, one has to be careful that the logic is *consistent*. The key disadvantage of using LF is that proofs are BIG.

- Up to 10x the size of the code!

Oracle-Based Proofs

Necula was able to compress these proofs quite a bit:

- The "proof" is represented as a stream of bits.
- At each proof step, at most k rules can match the goal.
- We read off $lg(k)$ bits to determine which rule to apply.

In essence, the bit-stream serves as an oracle, telling the checker which rule to apply next in the search for a proof.

In practice, this doesn't increase the TCB by much and is a significant win for both proof-size and proof-checking time.

In general, we could allow an arbitrary program to re-construct the proof. But then we have to worry about securing that program...

Fine-Grained Isolation

In TAL, we could specify "fine-grained" memory policies.

The set of addresses that are safe to access are determined by types.

For instance, if `v` : $\text{ptr}(\text{ptr}(\text{int},\text{int}))$ then we know it's safe to read `Mem[v]` as well as `Mem[Mem[v]]` and `Mem[Mem[v]+4]`.

So an obvious idea is to encode TAL types as assertions that can be used to prove safety.

Attempt

One could imagine introducing rules like this:

$$\frac{e : \mathsf{ptr}(\sigma_1; \tau; \sigma_2)}{\mathsf{addr}(e + \mathsf{sizeof}(\sigma_1))}$$

reflecting the elimination rules of TAL.

Problem

Suppose $\mathtt{r} : \text{ptr}(\text{ptr}(\text{int}))$ is our pre-condition and we execute:

```
   Mem[r] := 0xbad;
```

Then the post-condition is

$$\mathtt{r} : \text{ptr}(\text{ptr}(\text{int})) \wedge m = \text{upd}(x, \mathtt{r}, \mathtt{0xbad})!$$

So we might still conclude that `Mem[Mem[r]]` is a valid address!

The problem is that the predicate "$\mathtt{r} : \text{ptr}(\text{ptr}(\text{int}))$" should depend upon memory.

Second Attempt

We must index the typing relations by the memory in which they hold:

$$\frac{e :_m \mathsf{ptr}(\sigma_1; \tau; \sigma_2)}{\mathsf{addr}(e + \mathsf{sizeof}(\sigma_1))}$$

Note that now we can also conclude:

$$\frac{e :_m \mathsf{ptr}(\sigma_1; \tau; \sigma_2)}{\mathsf{sel}(m, e + \mathsf{sizeof}(\sigma_2)) :_m \tau}$$

Previous Problem

Now suppose $\mathtt{r} :_m \mathsf{ptr}(\mathsf{ptr}(\mathsf{int}))$ is our pre-condition and we execute:

```
Mem[r] := 0xbad;
```

Then the post-condition becomes

$$\mathtt{r} :_x \mathsf{ptr}(\mathsf{ptr}(\mathsf{int})) \land m = \mathsf{upd}(x, \mathtt{r}, \mathtt{0xbad})$$

So we can no longer prove that `Mem[Mem[r]]` is a valid address since `r` is a ptr(ptr(int)) only in the *old* memory.

Next Problem

Suppose our pre-condition is:

$$\texttt{r1} :_m \mathsf{ptr}(\mathsf{int}) \wedge \texttt{r2} :_m \mathsf{ptr}(\mathsf{int})$$

and we execute:

```
   Mem[r1] := 3;
```

Then the post-condition is:

$$\texttt{r1} :_x \mathsf{ptr}(\mathsf{int}) \wedge \texttt{r2} :_x \mathsf{ptr}(\mathsf{int}) \wedge m = \mathsf{upd}(x, \texttt{r1}, 3)$$

So if we attempt to dereference `r1` or `r2`, we can no longer show that they are valid addresses!

To Cut a Long Story Short...

It is possible to construct an appropriate set of proof rules, but all of the same problems arise as in TAL (e.g., type invariance, separating allocation & initialization, etc.)

It is also possible to *define* the types using higher-order logic.

- This is the approach taken by Appel's Foundational PCC project.

- The advantage is that the typing rules need not be trusted.

- Certain type-constructors (e.g., recurisive types) demand a fairly involved construction—see Appel & McAllister.

Optimization

VC-generation is extremely robust to *local* optimization.

- constant folding, constant and copy propagation, register assignment, dead-code elimination, common sub-expression elimination, ...

These transformations tend to preserve the post-condition modulo some relatively simple rewrites.

We could use the infrastructure to automatically prove that the optimized code is equivalent to the original.

More elaborate transformations (e.g., loop invariant removal) require adjusting the invariants attached to labels in non-trivial ways.

ML/Java to PCC

So putting the pieces together:

- Start with a type-safe, high-level language (e.g., ML or Java).
- Compile that to TAL using type-preserving transformations.
  - e.g., cps, closure, object conversion, etc.
- Compile the TAL to PCC:
  - The TAL label types give us the invariants that we need to do VC-generation for PCC.

– The TAL typing proof can be used to guide the construction of the VC proof.

Who constructs the proof?

The user does the hard work of finding the initial proof when she writes the code in a high-level language. The compiler just preserves the proof as it transforms the program.

The Future

Stepping back, what we've done is to achieve a very fine-grained isolation policy with a very minimal trusted computing base.

What if want to employ richer kinds of policies?

- e.g., availability
- e.g., confidentiality

One idea is to formulate a high-level language whose type (or proof) system lets us capture the relevant aspects of the policy and then compile that to a TAL/PCC, just as we did with memory isolation.

Confidentiality

There are emerging languages (c.f., Myers' Jif) that can enforce rich confidentiality (and dually, integrity) properties.

They augment types with *classifiers* (e.g., high, low).

The type system ensures that no high-security value influences a low-security value (non-interference.)

```
   low := hi mod 2;

   low := 0;
   if (hi mod 2 = 0) low := 1;
```

Notes

A big shortcoming of the current models, languages, and types is that they are too strong:

- we tend to get “label creep”.
- need support for declassification: `l := encrypt(h)`.
- compiler transformations (e.g., cps) often break typing.

Interestingly, the high/low-security labels can be encoded with just polymorphism (see Zdancewic et al., ICFP) and linearity helps avoid label creep.

Wide open: Jif to PCC.

Conclusions

Certifying compilation is starting to take off:

- Both Intel and Microsoft have serious development efforts.

The methodology works well when we leverage types.

- Users (and type inference) conspire to construct "proofs" at the source-level.

- Certifying compilers simply maintain them as they compile the code.

All of this suggests that we can practically enforce *much* stronger security policies without sacrificing performance or the size of the TCB.

Arrays

Extend memory types:

$$\text{memory types} \quad \sigma \quad ::= \quad \epsilon \mid \tau \mid \sigma_1 ; \sigma_2 \mid \sigma^e$$

with the understanding that:

$$\begin{aligned} \sigma^0 &= \epsilon \\ \sigma^{n+1} &= \sigma ; \sigma^n \end{aligned}$$

So an `int[42]` would become ptr(int^{42}).

Arrays Cont'd

Consider a function:

```
int f(int A[42], int y) {
    return A[y];
}
```

A type-safe compiler will insert bounds checks:

```
f: { A: ptr(int^42) ^ ...}
   if (y geq 42) goto Error;
   { A: ptr(int^42) ^ 0 ≤ y < 42 ^ ...}
   z := y*4;
   { z = sizeof(int^y) }
   z := A + z;
   { z : ptr(int^42 − y) }
   res := Mem[z];
   ...
```

Typed Assembly Language

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

June 14, 2007

References:

- G.Morrisett. Typed Assembly Language. In Benjamin C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, MIT Press, 2005.

- G.Morrisett, D.Walker, K.Crary, and N.Glew. From System-F to Typed Assembly Language, *ACM Transactions on Programming Languages and Systems*, 21(3):528-569, May 1999.

- See also http://www.cs.cornell.edu/talc

Type Safety Type-safe programming languages, such as Scheme, ML, and Java enjoy memory isolation.

- That is, type-safety implies memory isolation.

- In fact, it's a much stronger property.

Furthermore, type-safety is all about *fine-grained* memory safety.

- Having a value of type ptr(int) means you can safely read or write the address that value corresponds to.

- Having a value of type $\tau_1 \rightarrow \tau_2$ means you can safely jump to the address that value corresponds to.

And languages that are *statically* typed, such as ML and Java, don't have to pay the performance costs of masking or checking values.

Question: Why not force extensions to a service be written in ML or Java?

Performance?

- Ideally, we should beat the performance of SFI.
- That gives us a budget of 20% time overhead?
- In my experience, type-safe languages are much slower and fatter.
- Of course, we're enforcing a better policy...
- Key: if the overheads are too great, people won't use it.

Language-Independence 2. There are *many* type safe languages:

- SML, O'Caml, Haskell, Java, C#, Visual Basic, Mercury, Eiffel, Cyclone, Ccured, ...

- We should be able to pick a language suited to the task.

- A key advantage of OS/hardware and SFI is that they are language neutral.

Sun's JVM and Microsoft's .NET frameworks address these issues to some degree, but fall down in others:

- e.g., no support for tail-calls on the JVM
- e.g., improper treatment of array subtyping in both.
- forced object model.

Trust 3. Compilers and run-time systems for high-level, type-safe languages are complicated:

- SML/NJ: 100K lines of SML code, 50K lines of C code.
- It's very likely that the implementation has bugs.
- These points hold for the JVM and .NET worlds as well.

Typing Machine Code

- An alternative approach is to try to apply type-safety directly to *machine code*.
- That eliminates the need for a trusted compiler.
- Operationally, the machine code can efficidently support a wide variety of languages.
- The challenge is constructing a type system that is language-neutral.

TAL The goals of Typed Assembly Language (TAL):

- as in SFI, memory & control-flow safety
- as in SFI, try to be language-neutral
- unlike SFI, support “swiss cheese” (i.e., least priv.)
- unlike SFI, static enforcement

The reality:

- it’s not language-neutral
- (but it’s better than the JVM or CLR)
- See the web page for TALx86

Outline

- TAL-0: control-flow isolation
- TAL-1: polymorphism
- TAL-2: data isolation
- TAL-3: allocation and initialization
- TAL-4: data abstraction
- TAL Wrapup

TAL-0: control-flow isolation

TAL-0 Abstract Machine

$$
\begin{array}{rlcl}
\text{machines} & M & ::= & (H, R, I) \\
\text{heaps} & H & ::= & \{\ell_1 = I_1, \ldots, \ell_n = I_n\} \\
\text{register file} & R & ::= & \{r_1 = v_1, \ldots, r_k = v_k\} \\
\text{instr. sequences} & I & ::= & \text{jump } v \mid \iota; I
\end{array}
$$

- Register file is a total map from registers to (register-free) operands.

- Heap is a partial map from lables to instruction sequences (code).

- Machine state is a code heap, register file, & instruction sequence.

- I plays the role of a program counter.

Instruction Syntax

$$\begin{array}{rlll}
\text{registers} & r & ::= & r_1 \mid r_2 \mid \ldots \mid r_k \\
\text{integers} & n & & \\
\text{labels} & \ell & & \\
\text{operands} & v & ::= & r \mid n \mid \ell \\
\text{instructions} & \iota & ::= & r_d := v \qquad\qquad\qquad \text{(move)} \\
 & & \mid & r_d := r_s + v \qquad\quad\; \text{(add,addi)} \\
 & & \mid & \text{ifeq } r \text{ jump } v \qquad \text{(beq)}
\end{array}$$

Example A procedure that computes the product of the integers in `r1` and `r2`, placing the result in `r3` before jumping to the return address in `r4`.

```
prod: r3 := 0;          // res := 0
      jump loop
loop: if r1 jump done; // if a = 0 goto done
      r3 := r2 + r3;   // res := res + b
      r1 := r1 + -1;   // a := a - 1
      jump loop
done: jump r4          // return
```

Dynamics The rewriting rules for TAL-0 are as follows:

$$\frac{H(\hat{R}(v)) = I}{(H, R, \textsf{jump } v) \longrightarrow (H, R, I)}$$

$$(H, R, r_d := v; I) \longrightarrow (H, R[r_d = \hat{R}(v)], I)$$

$$\frac{R(r_s) = n_1 \qquad \hat{R}(v) = n_2}{(H, R, r_d := r_s + v; I) \longrightarrow (H, R[r_d = n_1 + n_2], I)}$$

$$\frac{R(r) = 0 \qquad H(\hat{R}(v)) = I'}{(H, R, \textsf{ifeq } r \textsf{ jump } v; I) \longrightarrow (H, R, I')}$$

$$\frac{R(r) = n \qquad n \neq 0}{(H, R, \textsf{ifeq } r \textsf{ jump } v; I) \longrightarrow (H, R, I)}$$

Comments

- Think of each instruction as a function/combinator from register files to register files.
- Semi-colon is the "bind" i.e., sequencing.
- Jump is the function call.
- All sequences end with jump and there's no nesting – CPS!
- It's easy to show a step-for-step simulation with a concrete MIPS semantics.

Policy No wild jumps (something we wanted for SFI).

- Positive re-statement: the only possible jumps are to the labelled instruction sequences.

- Baked into the formulation of the abstract machine!

- We get stuck if we try to jump to an integer or to an undefined label.

- Note that we also get stuck if we try to add a label and an integer.

- Moral: the policy is expressed by writing down an abstract machine. Configurations for which there is no transition are “bad”.

Statics Clearly, need to distinguish labels from integers.

$$\begin{array}{rcll} \text{types} & \tau & ::= & \mathsf{int} \mid \mathsf{code}(\Gamma) \\ \text{register file types} & \Gamma & ::= & \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\} \\ \text{heap types} & \Psi & ::= & \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\} \end{array}$$

- To keep the induction going, we need to know the types of the registers once we jump to a particular location (in case it jumps through a register.)

- code(Γ) describes a label which when jumped to, expects register r_i to have a value of type $\Gamma(r_i)$.

- Ψ describes a heap (i.e., association of code labels to their code types.)

Operands Typing Values & Operands: $\Psi \vdash v : \tau$, $\Psi; \Gamma \vdash v : \tau$:

$$\Psi \vdash n : \mathsf{int}$$

$$\Psi \vdash \ell : \Psi(\ell)$$

$$\frac{\Psi \vdash v : \tau}{\Psi; \Gamma \vdash v : \tau}$$

$$\Psi; \Gamma \vdash r : \Gamma(r)$$

Instructions Typing Instructions: $\Psi \vdash \iota : \Gamma_1 \rightarrow \Gamma_2$

$$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash r_d := v : \Gamma \rightarrow \Gamma[r_d : \tau]}$$

$$\frac{\Psi; \Gamma \vdash r_s : \mathsf{int} \qquad \Psi; \Gamma \vdash v : \mathsf{int}}{\Psi \vdash r_d := r_s + v : \Gamma \rightarrow \Gamma[r_d : \mathsf{int}]}$$

$$\frac{\Psi; \Gamma \vdash r_s : \mathsf{int} \qquad \Psi; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\Psi \vdash \mathsf{if}\ r_s\ \mathsf{jump}\ v : \Gamma \rightarrow \Gamma}$$

Sequences Typing Instruction Sequences: $\Psi \vdash I : \mathsf{code}(\Gamma)$

$$\frac{\Psi; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\Psi \vdash \mathsf{jump}\ v : \mathsf{code}(\Gamma)}$$

$$\frac{\Psi \vdash \iota : \Gamma \rightarrow \Gamma_2 \qquad \Psi \vdash I : \mathsf{code}(\Gamma_2)}{\Psi \vdash \iota; I : \mathsf{code}(\Gamma)}$$

Machines Typing Heaps, Register Files, & Machines:

$$\frac{\forall \ell \in Dom(\Psi).\Psi \vdash H(\ell) : \Psi(\ell)}{\vdash H : \Psi}$$

$$\frac{\forall r.\Psi \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma}$$

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I : \mathsf{code}(\Gamma)}{\Psi \vdash (H, R, I)}$$

Soundness

Theorem: if $\vdash M$ then M cannot become stuck (i.e., do a wild read/write/jump.) (i.e., $M \longmapsto^* M_1$ implies there exists an M_2 s.t. $M_1 \longmapsto M_2$.)

Proof: Show $\vdash M$ implies there exists an M' such that $M \longmapsto M'$ and $\vdash M'$. That is, show a well-typed program is not immediately stuck, and that well-typedness is an invariant under evaluation.

Proof Only one rule ends with $\vdash M$ so taking $M = (H, R, I)$, by inversion the derivation is of the form:

$$\frac{\dfrac{\mathcal{D}_H}{\vdash H : \Psi} \qquad \dfrac{\mathcal{D}_R}{\Psi \vdash R : \Gamma} \qquad \dfrac{\mathcal{D}_I}{\Psi \vdash I : \mathsf{code}(\Gamma)}}{\vdash (H, R, I)}$$

We proceed by cases on the last rule used in $\mathcal{D}_I$.

Proof, contd. Case: $\mathcal{D}_I$ ends with an instance of the jump rule:

$$\frac{\Psi; \Gamma \vdash v : \mathsf{code}(\Gamma)}{\Psi \vdash \mathsf{jump}\ v : \mathsf{code}(\Gamma)}$$

Lemma [Canonical Operands]: If $\Psi \vdash R : \Gamma$ and $\Psi; \Gamma \vdash v : \tau$ then:

1. if $\tau = \mathsf{int}$ then $\hat{R}(v) = i$ for some i, and
2. if $\tau = \mathsf{code}(\Gamma')$ then $\hat{R}(v) = \ell$ for some $\ell \in Dom(\Psi)$.

So from (2), we know $v = \ell \in Dom(\Psi)$.

From $\vdash H : \Psi$ we can conclude $H(\ell) = I'$ for some I' such that $\Psi \vdash I' : \text{code}(\Gamma)$.

So $(H, R, I) \overset{(}{\longmapsto} H, R, I')$. Furthermore, $\vdash (H, R, I')$.

Proof, contd. Case: $\mathcal{D}_I$ ends with an instance of the sequencing rule, where the instruction at the beginning is a move:

$$\frac{\dfrac{\Psi;\Gamma \vdash v : \tau}{\Psi;\Gamma \vdash r_d := v : \Gamma \to \Gamma[r_d{:}\tau]} \qquad \dfrac{\mathcal{D}}{\Psi \vdash I' : \mathsf{code}(\Gamma[r_d{:}\tau])}}{\Psi;\Gamma \vdash r_d := v; I' : \mathsf{code}(\Gamma)}$$

where $I = r_d := v; I'$.

Clearly, $(H, R, I) \longmapsto (H, R[r_d = \hat{R}(v)], I')$.

It's easy to show from $\Psi;\Gamma \vdash v : \tau$ and $\Psi \vdash R : \Gamma$ that $\Psi \vdash \hat{R}(v) : \tau$.

So it follows that $\Psi \vdash R[r_d = \hat{R}(v)] : \Gamma[r_d{:}\tau]$ and:

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R[r_d = \hat{R}(v)] : \Gamma[r_d{:}\tau] \qquad \Psi \vdash I' : \mathsf{code}(\Gamma[r_d{:}\tau])}{\vdash (H, R[r_d = \hat{R}(v)], I')}$$

Proof, contd. The other cases (where the instruction sequence starts with an add or a conditional branch) are similar.

Very easy proof.

However...

Oops! You can't write a program that jumps through a register! The only possible derivation for a jump through a register looks like this:

$$\dfrac{\dfrac{\Psi \vdash \Gamma(r) : \mathsf{code}(\Gamma)}{\Psi; \Gamma \vdash r : \mathsf{code}(\Gamma)}}{\Psi \vdash \mathsf{jump}\ r : \mathsf{code}(\Gamma)}$$

The top line means that we need $\Gamma(r) = \mathsf{code}(\Gamma)$ but there is no solution to this equation, given our inductively defined types.

Fixes There are lots of possible solutions to this "problem".

- Introduce recursive types ($\mu\alpha.\tau = \tau\{(\mu\alpha.\tau)/\alpha\}$).

- Introduce subtyping on register file types (e.g., code(Γ) $\leq$ int).

- Introduce universal polymorphism ($\forall\alpha.\tau$)

Adding all of these typing features is really necessary for a realistic TAL, but we can go a long way with just universal polymorphism.

TAL-1: polymorphism

Types revisited

$$\text{types}\ \ \tau\ \ ::=\ \ \text{int} \mid \text{code}(\Gamma) \mid \alpha \mid \forall\alpha.\tau$$

Values with type α must be treated abstractly.

Values with polymorphic types (e.g., $\forall\alpha.\tau$) can be instantiated by substituting a particular type for the bound type variable (e.g., $\tau\{\tau'/\alpha\}$. Formally:

$$\frac{\Psi;\Gamma \vdash v : \forall\alpha.\tau}{\Psi;\Gamma \vdash v : \tau\{\tau'/\alpha\}}$$

We can introduce polymorphic generalization only for instruction sequences:

$$\frac{\Psi \vdash I : \tau}{\Psi \vdash I : \forall \alpha . \tau}$$

- Polymorphism in the presence of mutable, shared data always causes trouble somewhere.

- By restricting generalization to code, we're avoiding some of the potential pitfalls (good rule of thumb.)

- We'll require that Ψ be closed so there will be no other free variables in the context.

Fixing the proof

Basically, introduce a bunch of substitution lemmas such as:

If $\Psi \vdash v : \forall\alpha.\tau$ then $\Psi \vdash v : \tau\{\tau'/\alpha\}$.

That's all there is to it!

How does it help? If the target is polymorphic in the register we're jumping through, then we can specialize it to the current code type.

Let $\tau = \forall\alpha.\mathsf{code}(\Gamma[r:\alpha])$.

$$\cfrac{\cfrac{\cfrac{\Psi \vdash \Gamma(r) : \tau}{\Psi \vdash \Gamma(r) : \mathsf{code}(\Gamma[r:\tau])}}{\Psi;\Gamma \vdash r : \mathsf{code}(\Gamma[r:\tau]))}}{\Psi \vdash \mathsf{jump}\ r : \mathsf{code}(\Gamma[r:\tau])}$$

In essence, we're substituting τ for α within $\mathsf{code}(\Gamma[r:\alpha])$ which is similar to what happens with recursive types.

Note that this requires *polymorphic recursion*—something SML does not support (but Haskell and O'Caml do.)

An example

```
prod: r3 := 0;              // res := 0
      jump loop
loop: if r1 jump done;      // if a = 0 goto done
      r3 := r2 + r3;        // res := res + b
      r1 := r1 + (-1);      // a := a - 1
      jump loop
done: jump r4               // return
```

- Let $\Gamma = \{r_1, r_2, r_3{:}\mathsf{int}, r_4{:}\forall\alpha.\mathsf{code}\{r_1, r_2, r_3{:}\mathsf{int}, r_4{:}\alpha\}\}$.
- Let Ψ map `prod`, `loop`, and `done` to code(Γ).

- Claim that Ψ is type-consistent with the code that is given.

Other Uses for Polymorphism Simulating subtyping:

- When compiling an "`if-then-else`", we need to jump to a common join-point after the `then`- and `else`-clauses.

- Suppose the `then`-clause moves code into `r1` whereas the `else`-clause moves an integer into `r1`.

- By making the join-point polymorphic in `r1` (e.g., $\forall\alpha.\text{code}\{r_1{:}\alpha,\ldots\}$) both paths can jump to the label. Here, α is treated as a "top".

- Another alternative would be to add union types (e.g., $\tau_1 \vee \tau_2$) or direct support for subtyping.

Callee-saves registers The advantage of universal polymorphism is that we can track input/output relationships. Consider a label ℓ with this type:

$$\ell : \forall\alpha.\text{code}\{r_1{:}\alpha,\ r_2{:}\text{code}\{r_1{:}\alpha, \ldots\}\ , \ldots\}$$

We can think of ℓ as a function that takes its argument in r_1 and its return address in r_2, and when it returns, it returns its result in r_1.

So, ℓ conceptually is the result of compiling something like: $\forall\alpha.\alpha \to \alpha$. That means that if we "call" ℓ, then it *must* return what we pass to it (if it returns at all.)

This is useful for capturing compiler idioms like callee-saves registers.

Moral You can go a long way with just ints, code, and universal polymorphism. And the proofs remain remarkably simple. But,

we still have a long way to go...

Checking vs. Inference The right way to conceptualize the code consumer is that we're passing it the entire proof that the program type-checks. The consumer is just verifying the proof and extracting the program.

This is a relatively easy (and hence trustworthy) process.

In contrast, giving you the code and asking you to *infer* whether it is typeable is likely to be undecidable.

However, the size of real "proofs" would be tremendous compared to the code. So we have to rethink the representation of the proof.

We could note that most of the proof rules are syntax directed and so they can be run backwards.

The notable exceptions are polymorphic generalization and instantiation, as well as “guessing” the right Ψ.

One solution (used in TALx86) is to introduce term constructs that witness which rule to pick when the rules are not syntax directed.

For instance, TALx86 had an explicit instantiation syntax for polymorphic values, and allowed explicit types to be put on the labels.

Drawbacks: types are still large, must trust reconstruction engine.

TAL-2: data isolation

Data Let's augment our abstract machine with data values:

$$\begin{array}{rrcl} \text{memory values} & m & ::= & I \mid \langle v_1, \ldots, v_n \rangle \\ \text{heaps} & H & ::= & \{\ell_1 = m_1, \ldots, \ell_n = m_n\} \end{array}$$

so now our memory (H) maps pointers to either code or sequences of word-sized values.

We could also imagine adding in load and store instructions:

$$\begin{array}{rrcll} \text{instructions} & \iota & ::= & \cdots & \\ & & \mid & r_d := \text{Mem}[r_s + i] & \text{load} \\ & & \mid & \text{Mem}[r_d + i] := r_s & \text{store} \end{array}$$

New Rules

$$\frac{R(r_s) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_j, \ldots, v_n \rangle \qquad 4 * j = i}{(H, R, r_d := \mathsf{Mem}[r_s + i]; I) \longrightarrow (H, R[r_d = v_j], I)}$$

$$\frac{R(r_d) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_j, \ldots, v_n \rangle \qquad 4 * j = i}{(H, R, \mathsf{Mem}[r_d + i] := r_s; I) \longrightarrow (H[\ell = \langle v_0, \ldots, R(r_s), \ldots, v_n \rangle], R, I)}$$

New Types

$$\begin{array}{llcl} \text{operand types} & \tau & ::= & \text{int} \mid \text{code}(\Gamma) \mid \text{ptr}(\sigma) \mid \alpha \mid \forall\alpha.\tau \mid \forall\rho.\tau \\ \text{memory types} & \sigma & ::= & \epsilon \mid \tau \mid \sigma_1;\sigma_2 \mid \rho \end{array}$$

- ptr(σ) is the type of a data pointer, where the data are described by σ.

- We'll treat σ's as equivalent modulo associativity of ";" with ϵ a left- and right-unit.

- Memory type variables (ρ) let us abstract over a whole sequence of types.

- We have to segregate abstraction over word-sized values (α) from abstraction over arbitrary-sized values (ρ).

Typing Load and Store

$$\frac{\Psi;\Gamma \vdash r_s : \mathsf{ptr}(\sigma_1;\tau;\sigma_2) \qquad \mathsf{sizeof}(\sigma) = i}{\Psi \vdash r_d := \mathsf{Mem}[r_s + i] : \Gamma \to \Gamma[r_d{:}\tau]}$$

$$\frac{\Psi;\Gamma \vdash r_d : \mathsf{ptr}(\sigma_1;\tau;\sigma_2) \qquad \mathsf{sizeof}(\sigma) = i \qquad \Psi;\Gamma \vdash r_s : \tau}{\Psi \vdash \mathsf{Mem}[r_d + i] := r_s : \Gamma \to \Gamma}$$

where

$$\mathsf{sizeof}(\mathsf{int}) = \mathsf{sizeof}(\mathsf{code}(\Gamma)) = \mathsf{sizeof}(\mathsf{ptr}(\sigma)) = \mathsf{sizeof}(\alpha) = 4$$
$$\mathsf{sizeof}(\sigma_1;\sigma_2) = \mathsf{sizeof}(\sigma_1) + \mathsf{sizeof}(\sigma_2)$$

Note, we need to statically know the size of σ_1 to validate the load, so it can't have a ρ within it.

Close... But alas, it falls short of what's needed:

- No way to allocate, initialize, or recycle memory.
  - But we need this for stack frames, records, objects, ...
- All loads and stores must use *constant* offsets relative to the base of an object.
  - This precludes arrays.
- There's no way to *refine* the type of a value.
  - Needed for datatypes, object downcasts, type-tests, etc.

TAL-3: allocation and initialization

High-Level Languages In a high-level language, such as ML or Java, the allocation and initialization of objects is in some sense *atomic*.

For instance, for an ML record {`x:int, f:int->int`}, the only way to create the record is to supply the initial values for `x` and `f` as in:

```
val r = {x = 3, f = (fn z => z + 1)}
```

This ensures that no one reads a field, thinking it's initialized, when it isn't.

But in TAL, we want to make the steps of allocation & initialization *explicit* to expose those steps to optimization (e.g., instruction scheduling, register allocation, tail-allocation, ...)

Allocation

To simplify things, let us add a new primitive instruction:

$$\text{instructions} \quad \iota \quad ::= \quad \cdots \mid \texttt{malloc}\, n$$

which allocates a fresh location ℓ to hold $n/4$ words, all initially zero:

$$(H, R, \texttt{malloc}\,(n*4); I) \rightarrow (H[\ell \mapsto \langle\underbrace{0, \ldots, 0}_{n}\rangle], R[r_1 \mapsto \ell], I)$$

$$(\ell \notin Dom(H))$$

The typing rule is then:

$$\Psi \vdash \texttt{malloc}\,(n*4) : \Gamma \rightarrow \Gamma[r_1{:}\mathsf{ptr}(\underbrace{\mathsf{int}, \cdots, \mathsf{int}}_{n})]$$

Example Now we can compile an ML expression, such as

```
val r = {x = 3, y = 42}
```

to the following TAL code:

```
malloc 8;              // r1 : ptr(int,int)
r := r1;               // r : ptr(int,int)
Mem[r+0] := 3;
Mem[r+4] := 42;
```

Oops! But this doesn't work when one of the components isn't an integer:

```
main:
  malloc 8;              // r1 : ptr(int,int)
  r := r1;               // r : ptr(int,int)
  Mem[r+0] := main;
  Mem[r+4] := 42;
```

because the rule for memory updates does not let us *change* the type of a location:

$$\frac{\Psi; \Gamma \vdash r_d : \mathsf{ptr}(\sigma_1; \tau; \sigma_2) \qquad \mathsf{sizeof}(\sigma) = i \qquad \Psi; \Gamma \vdash r_s : \tau}{\Psi \vdash \mathsf{Mem}[r_d + i] := r_s : \Gamma \to \Gamma}$$

Why not?

Consider:

```
let val x : (int->int) ref = ref (fn x => x)
    val y : (int->int) ref = x
in
   x := 3;
   (!y)(42)
end
```

Aha! High-level languages do not let you change the type of memory locations because a *reader* expects the types to remain the same.

Consider:

```
let val nil : ∀'a.'a list
    val r : ∀'a.('a list) ref  = ref nil
    val x : int list ref = r
    val y : (int->int) list ref = r
in
    x := [3];
    (hd(!y))(42);
end
```

Same problem—trying to use the same shared, updateable object at two different types.

Consider:

```
class Pt { int x,y; }
class Pt3d extends Pt { int z; }
void f(Pt3D[] A) {
  Pt[] B = A;  // allowed since Pt3D <= Pt
  B[0] = new Pt(1,2);
  A[0].z;     //  oops!
}
```

Same problem—trying to use the same shared, updateable object at two different types.

But what about Java variables?

```
class Pt { int x,y; }
class Pt3d extends Pt { int z; }
void f(Pt3D A) {
  Pt B = A;
  B = new Pt(1,2);
  A.z;
}
```

Nothing goes wrong here even though we changed the type of `B` from `Pt3D` to `Pt`. Hmmmm....

And recall our rule for updating registers:

$$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash r_d := v : \Gamma \rightarrow \Gamma[r_d : \tau]}$$

There's no requirement that the type of r_d stay the same.

So, there's something different about Java variables and TAL registers compared to ML, Java, or TAL references.

Principle The basic principle is:

If you're going to change the type of a location, then you need to change the types of all potential paths to that location (i.e., track aliasing precisely), or make sure that the change is compatible with all potential future accesses.

Principle You can instantiate this principle a number of ways:

1. Allow unknown aliasing, reading, writing $\Rightarrow$ type remains fixed.
2. Allow unknown read-only aliasing $\Rightarrow$ co-variant subtyping.
3. Allow unknown write-only aliasing $\Rightarrow$ contra-variant subtyping.
4. Don't allow aliasing, type can vary arbitrarily.

Option 4 is what we did with registers. Can we pull the same trick with memory?

Linearity Let us make a distinction between *unique* pointers (no aliasing) vs. normal pointers.

We will allow you to change the type of a unique pointer:

$$\frac{\begin{array}{c}\Psi;\Gamma \vdash r_d : \mathsf{uptr}(\sigma_1;\tau;\sigma_2) \\ \Psi;\Gamma \vdash r_s : \tau' \qquad \mathsf{sizeof}(\sigma_1) = i\end{array}}{\Psi \vdash \mathsf{Mem}[r_d + i] := r_s : \Gamma \rightarrow \Gamma[r_d : \mathsf{uptr}(\sigma_1;\tau';\sigma_2)]}$$

and we will modify `malloc`'s type to reflect that it returns a unique pointer:

$$\Psi \vdash \mathtt{malloc}\,(n * 4) : \Gamma \rightarrow \Gamma[r_1{:}\mathsf{uptr}(\underbrace{\mathsf{int}, \cdots, \mathsf{int}}_{n})]$$

Furthermore, we can always convert a unique pointer to an unrestricted pointer, as long as we give up the ability to change the type of the pointer.

We will make this explicit by adding an instruction to *commit* a unique pointer:

$$\frac{\Gamma(r) = \mathsf{uptr}(\sigma)}{\Psi \vdash \texttt{commit}\ r : \Gamma \to \Gamma[r : \mathsf{ptr}(\sigma)]}$$

Example For instance, the ML code:

```
val a = {x = 3, y = 42}
val b = {p = a, q = a}
```

could translate into:

```
malloc 8;          // r1:uptr(int,int)
r2 := r1;          // r2:uptr(int,int)
Mem[r2+0] := 3;    // r2:uptr(int,int)
Mem[r2+4] := 42;   // r2:uptr(int,int)
commit r2;         // r2:ptr(int,int)
malloc 8;          // r1:uptr(int,int)
Mem[r1+0] := r2;   // r1:uptr(ptr(int,int),int)
Mem[r1+4] := r2;   // r1:uptr(ptr(int,int),ptr(int,int))
commit r1;         // r1:ptr(ptr(int,int),ptr(int,int))
```

Good News/Bad News That's the good news. The bad news is that we have to maintain the uniqueness invariant.

The approach used is to destroy access to old copies. For instance, the move rule for unique pointers looks like:

$$\frac{\Gamma(r_s) = \mathsf{uptr}(\sigma)}{\Psi \vdash r_d := r_s : \Gamma \to \Gamma[r_d : \mathsf{uptr}(\sigma), r_s : \mathsf{int}]}$$

So, if you copy a unique pointer from r_s to r_d, then you can no longer use the copy in r_s.

Deallocation

We can also support an operation `free`:

$$(H[\ell \mapsto \langle v_1, \ldots, v_n \rangle], R[r \mapsto \ell], \texttt{free}\, r; I) \rightarrow (H, R[r \mapsto 0], I)$$

but we'd better restrict the pointer to be unique!

$$\frac{\Psi; \Gamma \vdash r : \mathsf{uptr}(\sigma)}{\Psi \vdash \texttt{free}\, r : \Gamma \rightarrow \Gamma[r : \mathsf{int}]}$$

Without this restriction, we could get stuck trying to dereference a dangling pointer!

That is, there's no guarantee that the program state remains closed.

Problems with free There are other problems with an explicit `free`:

- We may *recycle* the free'd location ℓ at a different type!
- That is, *free* locations can be captured by a subsequent `malloc`.
- This can lead to *extremely* subtle failures down the line.
- If the location contains unique pointers, then there's no way to regain access to them, resulting in a leak.

- In some sense, `malloc` and `free` are the ultimate in type-changing operations which is why they are fundamentally incompatible with type-safe languages.

- In turn, this tells you why garbage collection shows up in almost all type-safe languages.

Recycling

- Nonetheless, recycling of storage in a *controlled* fashion is important.
- In particular, most compilers want to recycle stack frames for different invokations of different procedures.
- It turns out that unique pointers are good enough for this as well.
- This is because we can think of `sp` as a register that holds a uptr(σ).

- As we’ll see, it’s very nice to have abstract memory types (ρ) to abstract the “tail” of the stack.

Example Example compilation:

```
int fact(int z) {
  if (z != 0) return prod(fact(z-1),z);
  else return 1;
}
```

Conventions:

- arguments are passed on the stack
- return address passed in `r4`

- results returned in `r1`
- callee pops arguments
- `r2` and `r3` are temps

But you can code your own convention if you like...

Type Translation A procedure with type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ maps down to:

```
$\forall \alpha, \beta, \gamma, \rho$.code{
      $r_1 : \alpha$   // don't care about value on input
      $r_2 : \beta$   // don't care about value on input
      $r_3 : \gamma$   // don't care about value on input
      $sp$ : uptr($\tau_1, \ldots, \tau_n, \rho$)   // $\tau_i$'s on top of stack, rest abstract ($\rho$)
      $r_4 : \forall \delta, \epsilon, \iota$.code{   // return addr.
            $r_1 : \tau$   // $\tau$ return value
            $r_2 : \delta$   // don't care about value on output
            $r_3 : \epsilon$   // don't care about value on output
            $sp$ : uptr($\rho$)   // arguments popped, rest intact
            $r_4 : \iota$   // abstract to support recursive type}}
```

Code Part I

```
fact:
   ∀a,b,c,s.
   code{r1:a,r2:b,r3:c,sp:uptr(int,s),
         r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,
                           sp:uptr(s)}}
   r1 := Mem[sp];     // r1:int, r1 := z
   if r1 jump retn    // if z = 0 goto retn
   r2 := r1 + (-1);   // r2:int, r2 := z-1
   salloc 2           // sp:uptr(int,int,int,s)
   Mem[sp+1] := r4;   // sp:uptr(int,(All ...),int,s)
   Mem[sp] := r2;     // sp:uptr(int,(All ...),int,s)
   r4 := cont;
   jump fact          // r1 := fact(z-1)
```

Code Part II

```
cont: ∀c,s’,d,e,f.
    code{r1:int,r2:d,r3:e,r4:f,
         sp:uptr(∀d,e,f.code{...},int,s’)}
   r4 := Mem[sp];   // restore original return address
   Mem[sp] := r1;   // sp:uptr(int,int,s’)
   jump prod        // tail call prod(fact(z-1),z)

retn: ∀b,c,s.
    code{r1:int,r2:b,r3:c,sp:uptr(int,s),
         r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,
                           sp:uptr(s)}}
   r1 := 1;
   sfree 1;       // sp:uptr(s)
   jump  r4       // return 1
```

Aside

However that this is only sufficient for dealing with very simple procedural control info, as there's no way to get a pointer into the middle of the stack.

That is, we can't support something like `&x`, so we can't handle stack allocation, frame pointers, static links, etc.

See the "Stack-Based Typed Assembly Language" (JFP) paper for some ways around this, or look at the work on region allocation.

TAL-4: data abstraction

Motivation If we're going to compile a modern OO or functional language, then we need support for objects and closures.

We could bake these types in, as the JVM and CLR do. But this forces you into a *particular* object/closure model.

Different languages and different compilers use wildly different encodings.

In the spirit of TAL, we should try to find more primitive type constructors that let us encode objects and closures...

Closures Let's start with closures first since they are much simpler.

In most compilers, closures are represented as a (pointer to a) pair of a piece of code and an environment.

The environment is a data structure that holds the values corresponding to the free variables in the code.

Closure Conversion For example, consider the ML source code:

```
val f : int -> (int -> int)
val f = fn x => let val g = fn y => x + y
                in g
                end
```

After closure conversion, the code might look like this:

```
gc(genv:{fenv=unit,x:int},y:int) = genv.x + y

fc(fenv:unit, y:int) =
  let val genv = {fenv=fenv, x=x}
      val g = {code=gc, env=genv}
  in g
  end
```

Closure Conversion In general, a function of type $\tau \to \tau'$ with free variables $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ will be translated into a pair $\{\text{code}, \text{env}\}$ where:

- env is a record of type $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$.

- code is a code pointer that takes two arguments, the first of which is `env` and the second of type τ', and returns a τ'.

So at the TAL-level $\tau \to \tau'$ becomes something like:

$$\text{ptr}((\text{code}(\tau_{\text{env}}, \tau) \Rightarrow \tau'), \tau_{\text{env}})$$

where "$\Rightarrow$" abstracts the calling convention of the generated code.

Oops! Different $\tau \to \tau'$ values will have different environments.

```
let val f1 = fn s:string => fn x:int => length(s)+x
    val f2 = fn z:int => fn y:int => z+y
in
   if flip() then
     (f1 "foo")
   else
     (f2 42)
end
```

But they need to have the same type to get a compositional translation.

Existential Types The solution is to use an *existential* to abstract the type of the environment:

$$\mathcal{T}[\![\tau \to \tau']\!] = \exists\alpha.\mathsf{ptr}((\mathsf{code}(\alpha, \tau) \Rightarrow \tau'), \alpha)$$

Existentials are introduced and eliminated as follows:

$$\frac{\Psi; \Gamma \vdash v : \tau[\tau'/\alpha]}{\Psi; \Gamma \vdash v : \exists\alpha.\tau}$$

$$\frac{\Psi; \Gamma \vdash r : \exists\alpha.\tau}{\Psi \vdash \mathtt{unpack}\, r : \Gamma \to \Gamma[r : \tau]}(\alpha \text{ fresh})$$

The freshness constraint ensures that we treat α abstractly.

Application So an application of a closure `f` of type int → int to an argument 42 might get compiled to something like this:

```
                      // r1 : ∃ a.(code(a,int)=>int, a)
   unpack r1;         // r1 : (code(b,int)=>int, b)
   r4 := Mem[r1+0];   // get out code of closure
   renv := Mem[r2+4]; // get out environment
   ra := cont;        // set the return address to cont
   r1 := 42;          // put argument in r2
   jump r4;           // call code
cont: ...             // return here.
```

Closures as Objects Closures are in fact a degenerate case of objects:

$$\mathcal{T}[\![\tau \to \tau']\!] = \exists\alpha.\mathsf{ptr}((\mathsf{code}(\alpha,\tau) \Rightarrow \tau'),\alpha)$$

We can generalize this to:

$$\exists\rho_1,\rho_2.\mathsf{ptr}(\mathsf{ptr}(\sigma_{vt},\rho_1),\tau_1,\ldots,\tau_n,\rho_2)$$

where σ_{vt},ρ_1 is a sequence of code pointers representing the method table.

Here, $\tau_1,\ldots,\tau_n$ correspond to "public" fields (accessible outside the object) and ρ_2 corresponds to "private" fields, (accessible only to the methods of the object.)

Similarly, ρ_1 abstracts a super-class's methods, so they are available within the methods, but not outside.

Example Ignore vtables for a second and just consider instance variables for two classes:

```
class Pt { int x,y; }
class Pt3d extends Pt { int z; }
```

Concretely, they would map down to:

- `Pt` = ptr(int, int)
- `Pt3D` = ptr(int, int, int)

We want to be able to pass a `Pt3D` (or any future extension of `Pt`) to functions that expect a `Pt`. Both can be abstracted to:

$$\exists\rho.\mathsf{ptr}(\mathsf{int}, \mathsf{int}, \rho)$$

For `Pt`, we pick $\rho = \epsilon$ and for `Pt3D`, we pick $\rho = \mathsf{int}$.

Object Encodings This style of encoding:

$$\exists\rho_1,\rho_2.\mathsf{ptr}(\mathsf{ptr}(\sigma_{vt},\rho_1),\tau_1,\ldots,\tau_n,\rho_2)$$

is similar to Pierce & Turner's "objects as existentials".

- In practice, we need to also add support for recursive types (and closures) to support the "self" parameter within methods (see Bruce.)

- Using this style of encoding, we can compile both Java-style OO languages as well as (call-by-value) functional languages such as ML.

- As with calling conventions, we are not forcing a particular representation on the compiler.

Recap

- Aliasing: the root of all evil.
- Uniqueness (a.k.a. linearity) provides convenient power-to-weight.
- Abstraction ($\forall$ and $\exists$) gets us surprisingly far.
- At this point, we can compile mini-Java and mini-ML.

Beyond... People like Karl Crary, Zhong Shao, and I have worked to develop more complete TAL's that scale up to realistic languages.

- arrays: need support for reasoning about *dynamic* offsets in loads and stores. See DTAL by Xi & Harper.

- datatype tests: need support for refining the type of a disjoint union based on a run-time test. See TALTwo by Crary.

- downcasts: need support similar to the above. See Neal Glew's thesis.

- exceptions: need support for stack-unwinding. See STAL.

The resulting type systems are fairly complicated. Should we trust them?

Up next: Proof-carrying code.

A Type-Safe Dialect of C

Greg Morrisett

Harvard University

Collaborators: D.Grossman, T.Jim, M.Hicks

C is a *terrible* language:

- Must bypass the type system to do simple things (e.g., allocate and initialize an object).
- Libraries put the onus on the client to do the “right thing” (e.g., check return codes, allocate data of right size, pass in array sizes, etc.).
- Manual memory management leads to leaks, data corruption.
- No information at runtime to do needed checks.(e.g., printf is passed arguments of the right type).
- "Portability" is in the #ifdef's, #defines, and Makefiles.

But C Is Also Very Useful:

Almost every critical system is coded in C:

- ported to lots of architectures.
- low-level control over data structures, memory management, instructions, etc.
- features useful for building device derivers, operating systems, protocol stacks, language runtimes, etc.
- the portability of the world is encoded in .h files

Questions:

- How do we achieve type safety for legacy C code?
- What should a next-generation C look like?

A Number of Recent Projects:

- LCLint, Splint [Evans]
- ESC M3/Java [Leino et al.]
- Prefix, Prefast [MS]
- SLAM [Ball, Rajamani]
- ESP [Das, Adams, Jagannathan]
- Vault, Fugue [Fahndrich, DeLine]
- Metal [Engler]
- CCured [Necula]

General Flavor

- Find bugs & inconsistencies in *real* source code.
  - e.g., Windows, Linux, Office, GCC, etc.
  - buffer overruns, tainted input, protocol violations, etc.
- A variety of analysis techniques.
  - ast analysis, dataflow analysis, type inference, constraint solving, model checking, theorem proving, spell checking,...
- Key needs:
  - minimize "false positives"
    - tool won't be used if it's not finding real bugs.
    - skip soundness, add annotations, add run-time checks, etc.
  - attention to scale
    - modular analysis, avoiding state explosion, etc.
  - good user interface
    - e.g., minimal error traces, integration with build system, etc.

The Cyclone Project

Cyclone is a type-safe dialect of C:

- primary goal: guarantee *fail-stop* behavior.
  - if we can't verify statically, we verify it dynamically.
  - whether or not we issue a warning is heuristic.
- second goal: retain virtues of C
  - syntax and semantics in the spirit of the language.
  - avoid hidden state (i.e., type tags, array bounds).
  - make it easy to interoperate with C (e.g., <kernel.h>).
  - ultimately: attractive for writing systems code.
- final goal: keep verification modular and scalable.
  - want this to be used as part of *every* build.
  - local analysis and inference only.
  - defaults, porting tool to minimize annotation burden.

Cyclone Users

- In-kernel Network Monitoring [Penn]
- MediaNet [Maryland & Cornell]
- Open Kernel Environment [Leiden]
- RBClick Router [Utah]
- xTCP [Utah & Washington]
- Lego Mindstorm on BrickOS [Utah]
- Cyclone on Nintendo DS
- Cyclone compiler, tools, & libraries
  - Over 100 KLOC
  - Plus many sample apps, benchmarks, etc.
  - Good to eat your own dog food…

This Talk

- A little bit about the Cyclone design:
  - Refining C types
  - Flow analysis
  - Type-safe Manual Memory management
- Lessons learned:
  - Theory vs. Practice
  - Why you shouldn’t trust tools
- Where we’re heading:
  - Open, trustworthy analysis framework

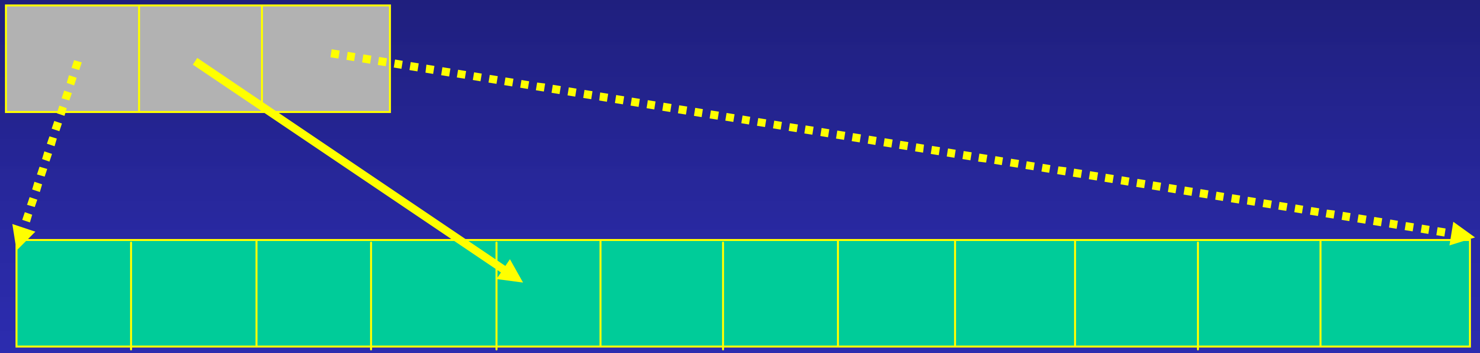
Hello World in Cyclone

```
#include <stdio.h>

int main(int argc, char*@zeroterm *@fat argv)
{
  if (argc != 2) {
    fprintf(stderr,"usage: %s <name>\n",argv[0]);
    exit(-1);
  }
  printf("Hello, %s.\n",*(++argv));
  return 0;
}
```

Fat Pointers:

To support dynamic checks, we must insert extra information (e.g., bounds for an array):



This is similar to what's done in Java, but we need more information to support pointer arithmetic.

Avoiding Overheads:

Dynamic checks make porting from C easy and our static analysis eliminates much of the overhead.

But often programmers want to *ensure* there will be no overhead and no potential failure.

To achieve this, programmers can leverage Cyclone's *refined types* and *static assertions.*

Pointer Qualifiers Clarify

Thin pointers: same representation as C, but restrictions on pointer arithmetic.

`char *`: a (possibly NULL) pointer to a character.
`char *@notnull`: a (definitely not NULL) pointer to a character.
`char *@numelts{c}`: pointer to a sequence of *c* characters.
`char *@zeroterm` : pointer to a zero-terminated sequence.

Fat pointers: arbitrary arithmetic but the representation is different (3 words):

`char *@fat` : a "fat" pointer to a sequence of characters.
`numelts(s)` : returns number of elements in sequence s

Subtyping Is Crucial:

Some Subtyping:

```
@numelts{42} <= @numelts{3}
@notnull <= @nullable
(mutable) <= @const
```

Some *No-check* Coercions:

```
@thin @numelts{42} <:= @fat
@thin @zeroterm <:= @fat @zeroterm
@fat @zeroterm <:= const @fat @nozeroterm
```

Some *Checked* Coercions:

```
@fat <#= @numelts{42}
@nullable <#= @notnull
```

Determining Qualifiers

Programmers:

- provide qualifiers for procedure interfaces

Compiler:

- infers qualifiers for local variables using a constraint-based inference algorithm.
- inserts coercions to adjust where necessary and possible.
- emits warnings for (most) checked coercions.

Porting Tool:

- global analysis tries to infer qualifiers, using only equality constraints (linear time).
- may be unsound(!) but compiler will flag problems

Checked Coercions & Warnings

In Cyclone stdio library:

```
FILE* fopen(const char *,const char *);
int getc(FILE *@notnull);
```

A client of the library:

```
FILE *f = fopen("foo.txt", "r");
c = getc(f);
```

Warning: argument might be NULL – inserting runtime check

Should Be Able to Avoid Warnings:

1. `cyclone -nowarn`

2.
   ```
   FILE @f = (FILE @)fopen("foo.txt", "r");
   c = getc(f)
   ```

3.
   ```
   FILE *f = fopen("foo.txt", "r");
   if (f == NULL) {
     perror("cannot find foo.txt\n");
     exit(-1);
   }
   c = getc(f)
   ```

Flow Analysis

Simple intraprocedural flow-sensitive, path-insensitive analysis used to determine:

- whether pointer variables are NULL.
  - used to avoid NULL checks, warnings.
- whether variables and fields within data structures are initialized.
  - warning on "bits-only" types, error otherwise.
- unsigned integer inequalities on variables.
  - used to avoid bounds checks, warnings.
- aliasing (essentially k-level with k = 2).
- "noreturn" attribute (e.g., calls exit).

An Example:

```
int strcmp(const char *@fat s1,
           const char *@fat s2) {
  unsigned n1 = numelts(s1);
  unsigned n2 = numelts(s2);
  unsigned n = (n1 <= n2) ? n1 : n2;
  for (int i = 0; i < n; i++) {
    ... s1[i] ... s2[i] ...
  }
    ...
}
```

The analysis is not able to prove that `i` is in bounds, so it inserts run-time tests...

Using Static Asserts

```
int strcmp(const char *@fat s1,
           const char *@fat s2) {
  unsigned n1 = numelts(s1);
  unsigned n2 = numelts(s2);
  unsigned n = (n1 <= n2) ? n1 : n2;
  @assert(n <= n1 && n <= n2);
  for (int i = 0; i < n; i++) {
    ... s1[i] ... s2[i] ...
  }
    ...
}
```

Here, we have

`n1 == numelts(s1) &`

`n <= n1 &  i < n`

In Practice:

- Initial code has lots of dynamic checks.
  - Choice of warning levels reveals *likely* points of failure.
- Two options:
  - Turn up knob on analyses
    - e.g., explore up to K paths
  - Refine types, add assertions
    - Programmer intensive

In either case, programmer views task as *optimizing* code when in fact, they're providing the important bits of a proof of safety.

One Big Wrinkle: Order

Order of evaluation is not specified for many compound expressions.

Consider: $e(e_1,e_2,\ldots,e_n)$

- Worst case: compiler could evaluate each expression in parallel.
- Even if you assume compiler does some permutation, you still have (n+1)! orderings.
- Could calculate all flows and then join, but that's too expensive in practice.

Solutions:

Originally, we had a sophisticated, iterative analysis to deal with the ordering issue.

- Complicated, difficult to maintain.

Now we force the order of evaluation.

- Greatly simplifies the analysis.
- Very little perceived loss in performance.
- But confuses GCC in some instances (e.g., self-tail calls.)

Moral: shouldn't be afraid to change the language to suit verification task.

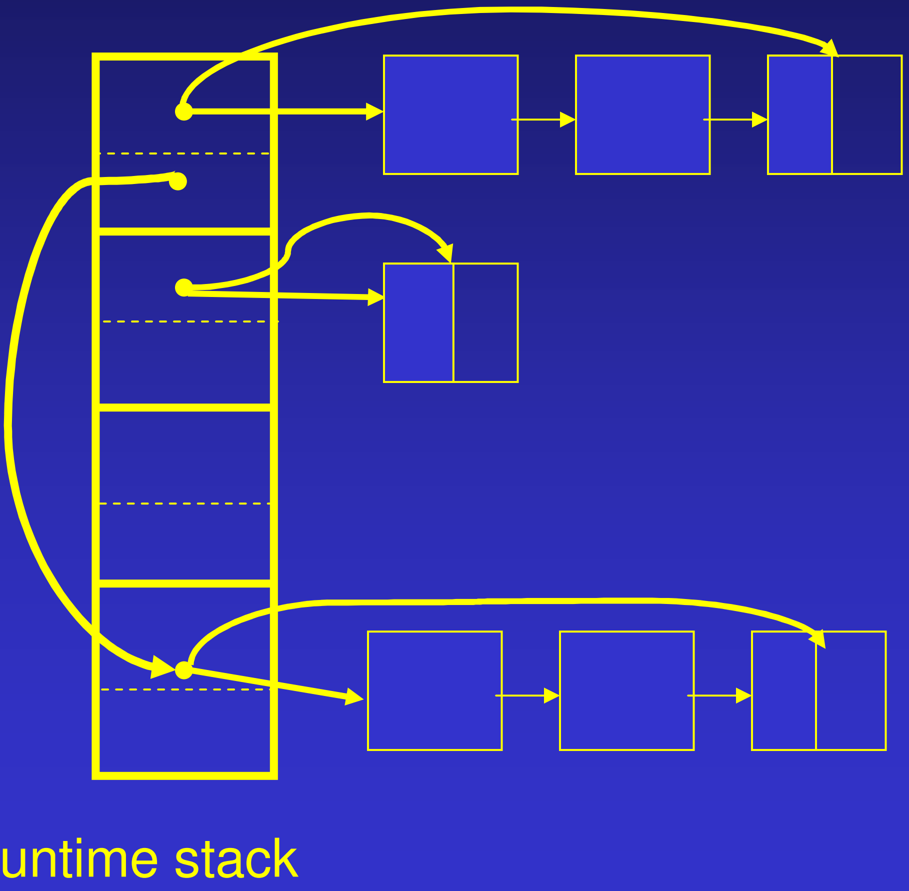
Other Cyclone Features:

- Unions
  - `union Foo {int x; float y;};`
    - can read or write either element
  - `union Bar {int *x; float y;};`
    - can write either element, but only read float
  - `@tagged union Baz {int *x; float y;};`
    - can read/write, but extra tag is inserted
- Parametric Polymorphism, Pattern-Matching, Existential Types, and Exceptions
- Limited dependent types over integer expressions (*a la* Dependent ML)
- Region-based memory management.

Region Goals

- Provide some mechanism to avoid GC.
  - no hidden tags.
  - no hidden pauses.
  - small run-time.
  - but ensure safe pointer dereferences.
  - scalable and modular analysis.
- Regions (a la Tofte & Talpin) fit the bill.
  - group objects with similar lifetimes into regions.
  - put region names on pointer types (`int *`r`).
  - track whether or not a region is live (effects).
  - allow dereferencing a pointer only if region is live.

Runtime Organization



Regions are linked lists of pages.

Arbitrary inter-region references.

Similar to arena-style allocators.

The Good News

Stack allocation happens a lot in C code.

- Thread local
- Cheap

Lexical region allocation works well for:

- "callee" allocates idioms (e.g., rgets)
- temporary data (e.g., environments)

Automatic deallocation.

All checks are done statically.

Real-time memory management.

The Bad News:

LIFO region lifetimes are too strict.

- No “tail-call” for regions.
- Lifetimes must be statically determined.
- Consider a server that creates some object upon a request, and only deallocates that object upon a subsequent request…

Creating/destroying a region is relatively expensive compared to malloc/free.

- Must install exception handler.
- Makes sense only when you can amortize costs over many objects.

To Address Shortcomings

- Unique pointers
  - Lightweight when compared to a region.
  - Can deallocate (free) at will.
  - But you can’t make a copy of the pointer.
- Dynamic regions
  - Can allocate or deallocate the arena at will.
  - Use a unique pointer as a “key” for access.

The combination actually subsumes lexical regions and provides the flexibility needed to optimize memory management for clients.

The Flexibility Pays: MediaNET

TTCP benchmark (packet forwarding):

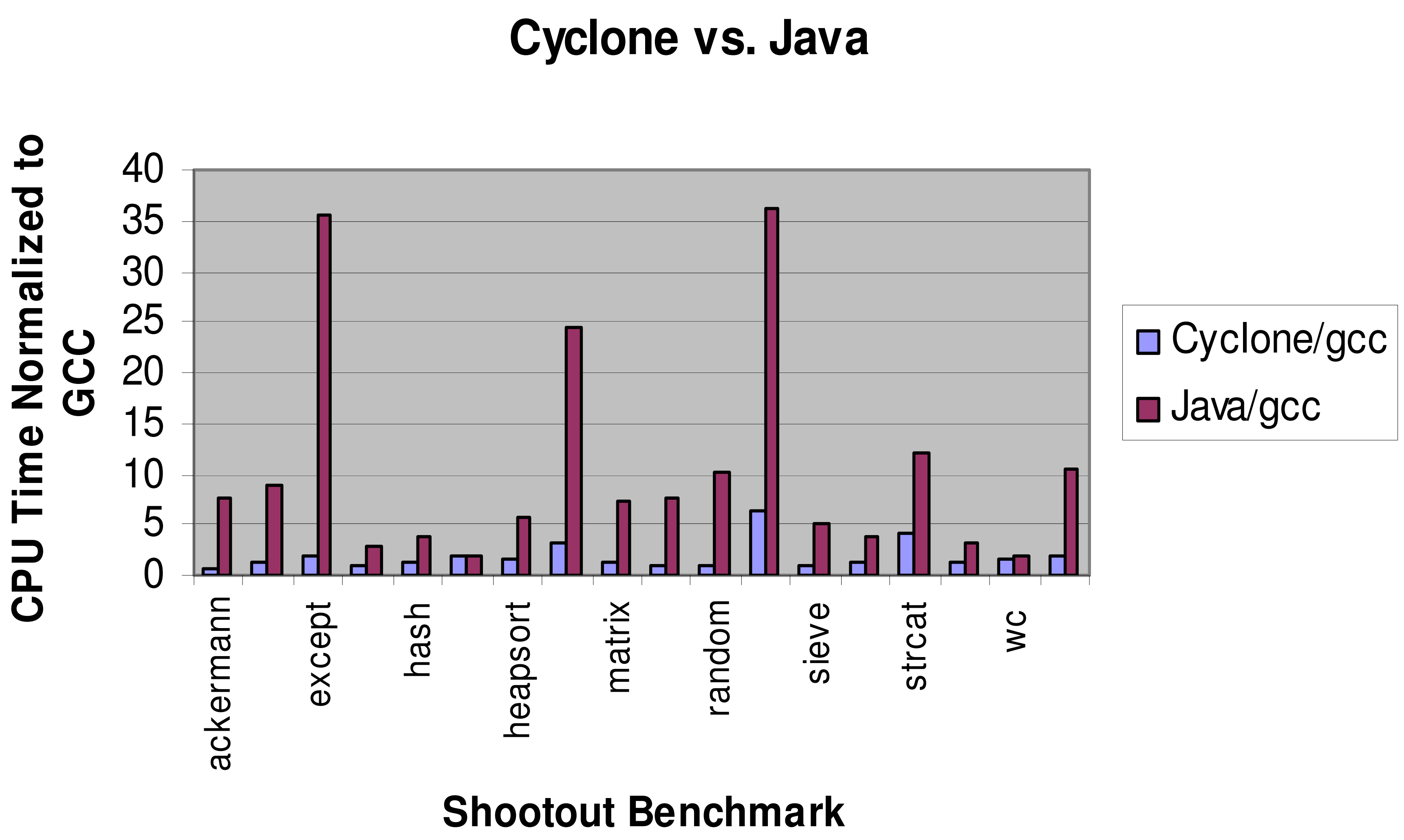
Cyclone v.0.1 (lexical regions & BDW GC)

- High water mark: 840 KB
- 130 collections
- Basic throughput: 50 MB/s

Cyclone v.0.5 (unique ptrs + dynamic regions)

- High water mark: 8 KB
- 0 collections
- Basic throughput: 74MB/s

Cyclone vs. Java



Comparing to Java

Program	Cyclone/gcc	Java/gcc
Ackermann	0.75	7.57
Ary3	1.21	8.85
Except	2.02	35.45
Fibo	1.00	2.86
Hash	1.35	3.83
Hash2	1.80	1.82
Heapsort	1.58	5.84
Lists	3.04	24.33
Matrix	1.24	7.30
Nestedloop	0.99	7.72
Random	0.99	10.11
Reversefile	6.45	36.28
Sieve	0.99	5.17
Spellcheck	1.15	3.67
Strcat	4.22	12.00
Sumcol	1.20	3.21
Wc	1.73	2.02

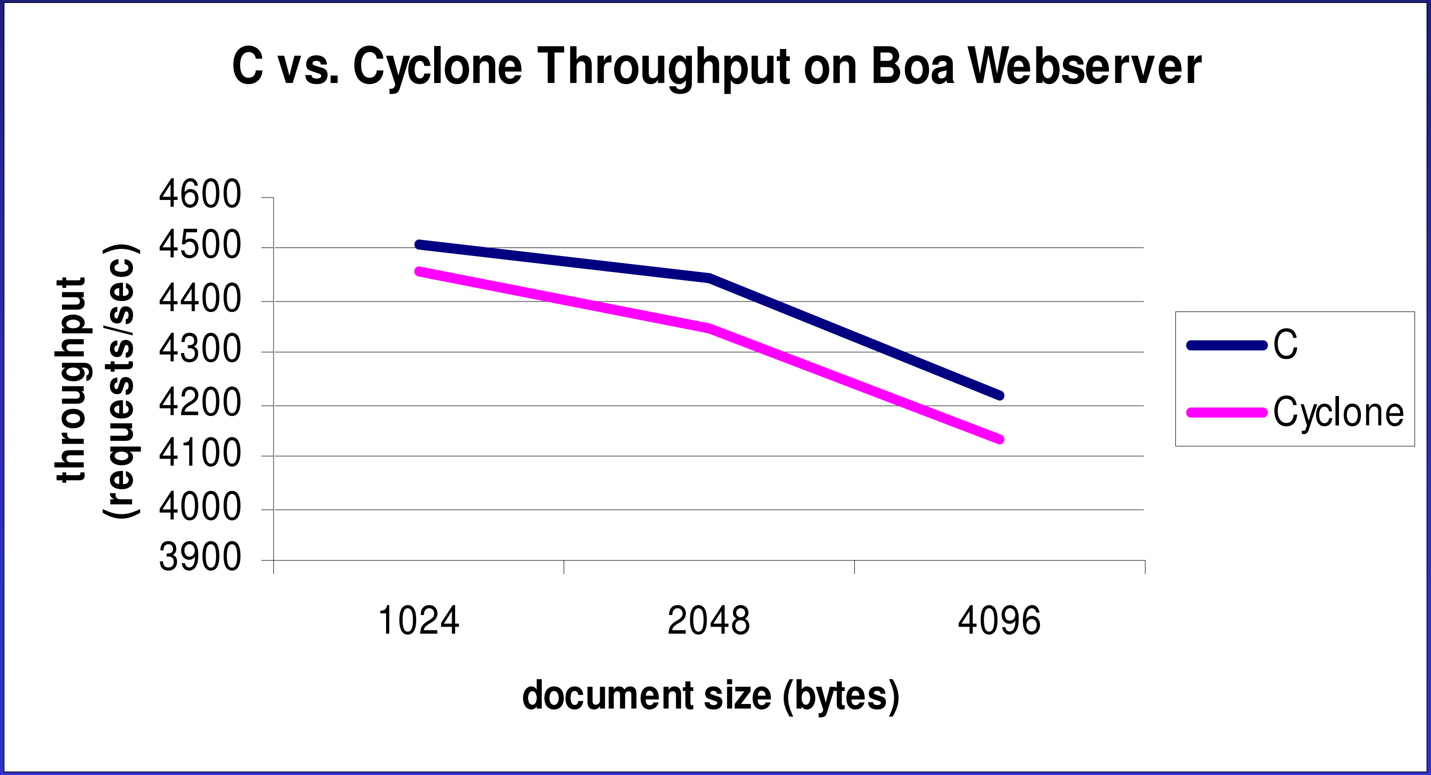
Bagley's Language Shootout comparing Sun's Java 2 RTE v1.4.1_03-b02.

CPU time normalized to gcc's.

On average:
Cyclone: 1.87
Java : 10.47

Macro-benchmarks:

We have also ported a variety of security-critical applications where we see little overhead (e.g., 2% for the Boa Webserver.)



Some Lessons Learned

- Don’t try to “fix” C:
  - Example: auto-break in switch cases
  - Instead, explicit “fallthru” annotation.
- There is no ANSI C:
- People matter, performance doesn’t
  - Porting code is still too painful.
  - Error messages are crucial.
- Interoperability is crucial.

Very Important Lessons

The compiler at this point is huge:

- ~ 50KLOC
- We kept finding subtle bugs in the analyses (c.f., order of evaluation.)
- Is it trustworthy?

Furthermore, there's no end to the refinements needed.

- Can we simplify the approach?

Current Thrust:

We're currently working on a more trustworthy, extensible infrastructure:

- As in ESC and SPLint:
  - Compiler computes verification conditions (using strongest-post-conditions.)
  - Infers some minimal loop invariants, but programmers can supply better invariants.
  - Uses an internal theorem prover to discharge most of the VCs.
- Unlike ESC/SPLint:
  - The prover is *not* trusted: must give witness.
  - If we can't prove it, then we do the run-time check.

Longer Term:

No need to stick with our prover:

- Should be able to discharge VCs using any plug-in prover, as long as it can produce a witness that we can check.
- In fact, should be able to discharge some proofs by hand!

Problem:

- Very few sound, witness-producing provers with useful decision procedures.
- For instance, few of them deal with machine arithmetic, and those that do don’t scale well.

The Program Logic

Another issue is fixing the logic to deal with issues such as memory mgmt.

The usual encoding of memory as a big array is insufficient for many reasons.

Hoping to leverage the emerging spatial logics (e.g., Reynolds & Ohearn's BI).

Open question: decision procedures.

Summary:

Cyclone is a type-safe dialect of C

- *Much* better performance than previous type-safe languages.
- In large part because programmers can tune performance (erm, safety) by adding additional information.
- More suited to writing new systems code than porting legacy code.
- Our ultimate goal is to make it possible (but not necessary) to eliminate all run-time checks.

More info...

`www.eecs.harvard.edu/~greg/Cyclone`