A Single Intermediate Language That Supports Multiple Implementations of Exceptions

Norman Ramsey, Harvard
Simon Peyton Jones, Microsoft

`http://www.cminusminus.org`

So you want to write a compiler

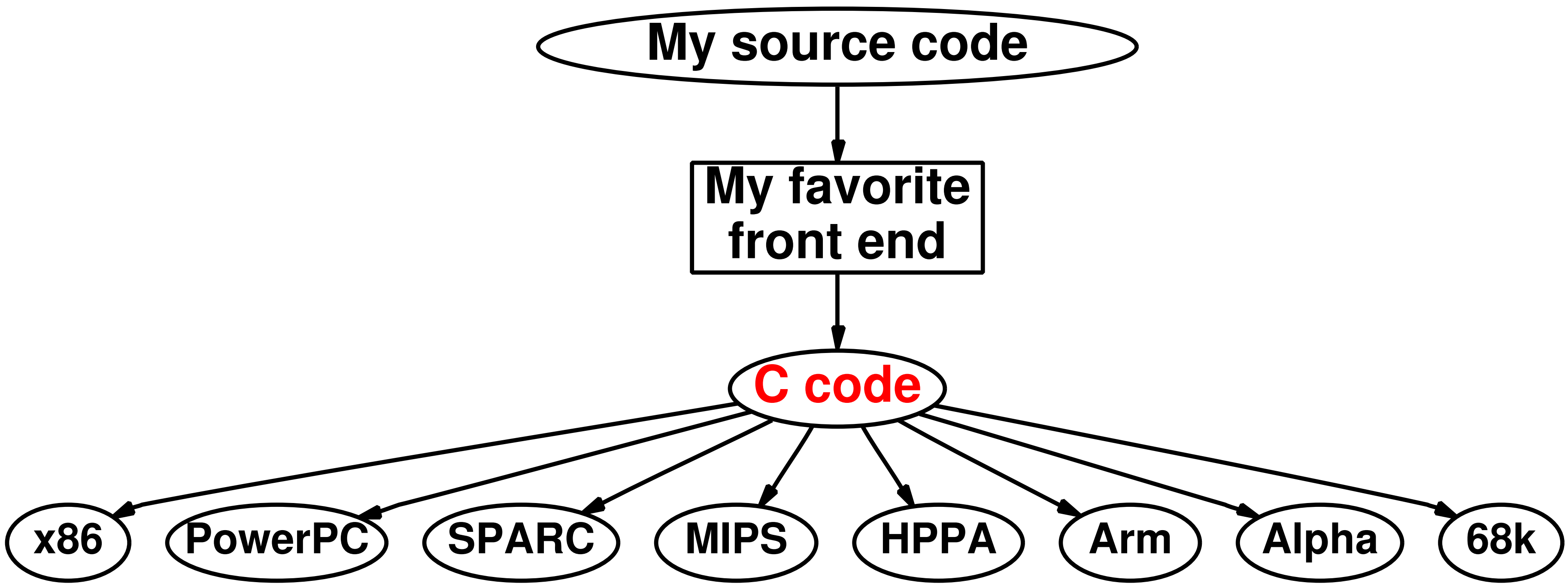
Enormous knowledge of compiler technology…

but a big job to implement

- **SSA, PRE, inlining, register coalescing…**
- **SPARC, MIPS, Alpha, HP-PA, PowerPC, 68020, Pentium…**

Why not look for a shortcut?

The garden path



The awful truth

Not quite as good as your own code generator:

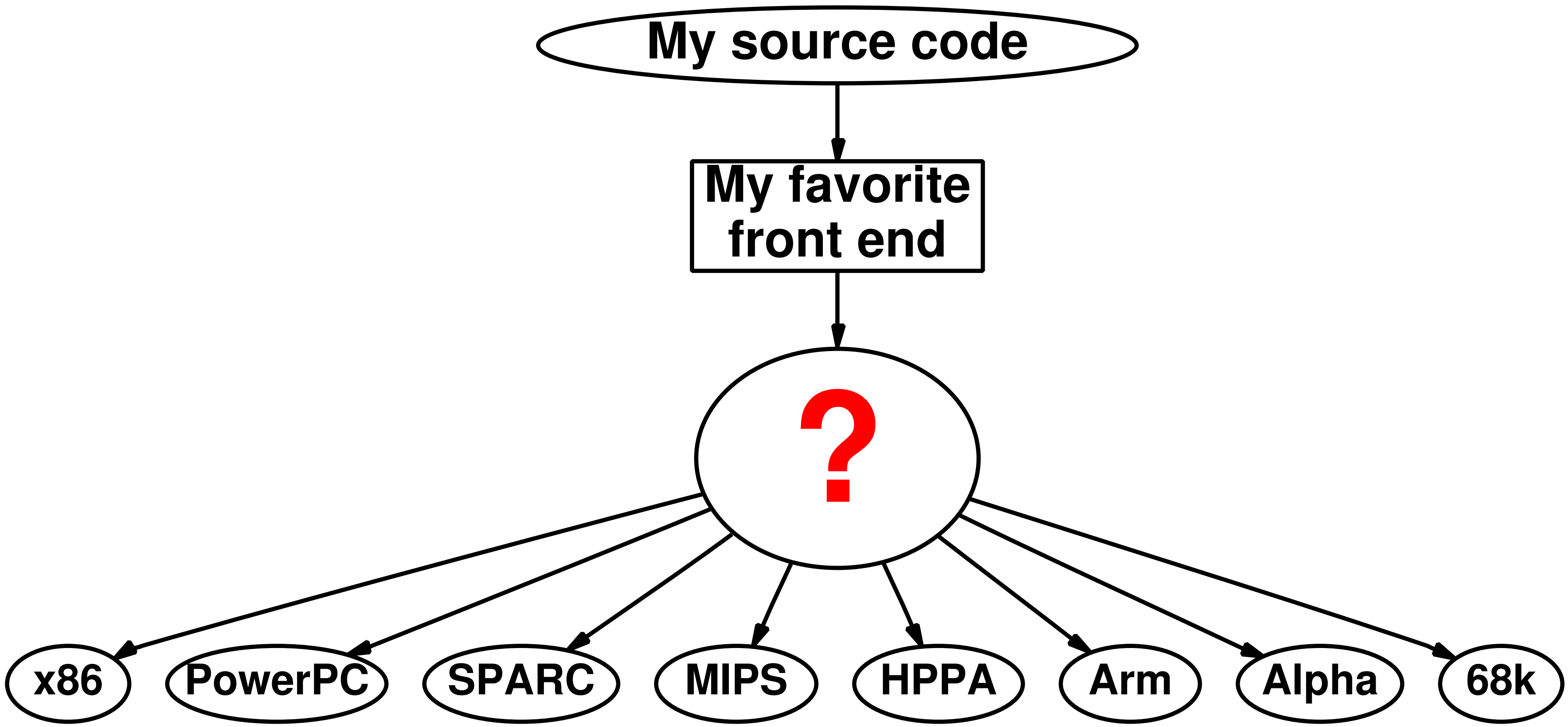
- **No proper tail calls**
- **Pointers may alias**
- **Can't return multiple registers**
- **No accurate garbage collection**
- `setjmp` **and** `longjmp` **for exceptions**
- **Forget source-level debugging**

Alternative—A compiler infrastructure?

Retargetable code generators exist

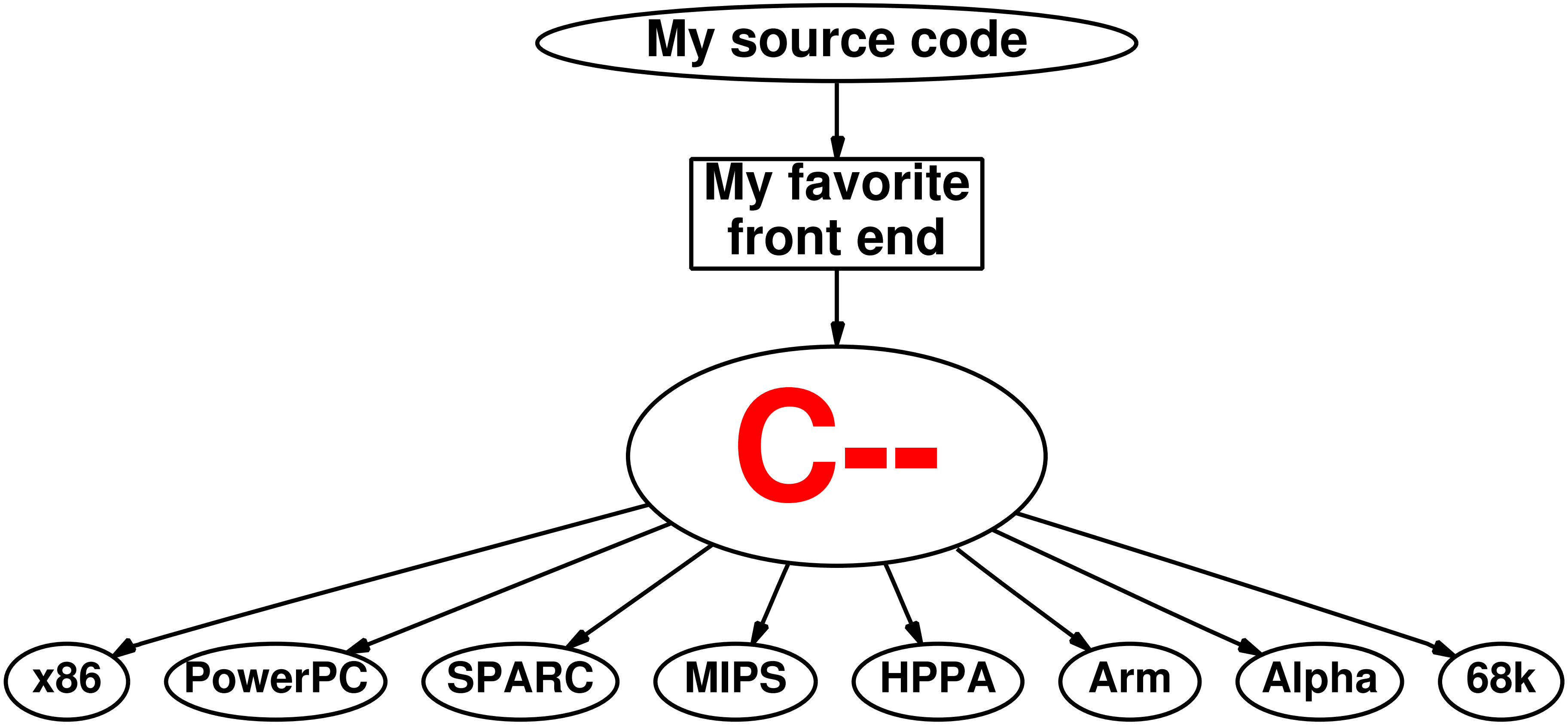
- **High entry cost**
  (big, complicated, poor documentation)
- **May push an implementation language**
  (gcc $\rightarrow$ C, MLRISC $\rightarrow$ ML)
- **May be inflexible**
  (vpo: no tail calls, C call conventions)

A nice model



My source code
My favorite front end
?
x86
PowerPC
SPARC
MIPS
HPPA
Arm
Alpha
68k

A new language



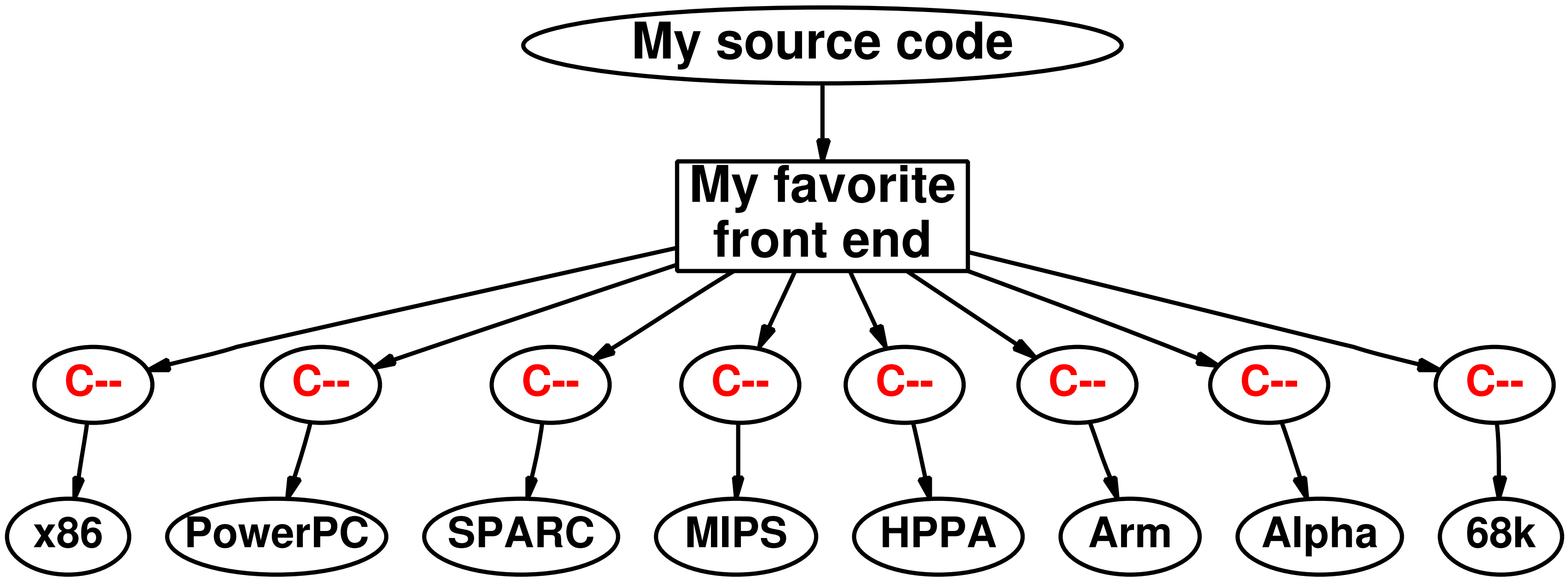
Example: array min and max

```
minmaxhelp(bits32 elements, bits32 limit,
           bits32 min, bits32 max)
{
  bits32 x;
  if (elements == limit) {
    return (min, max);
  }
  x = bits32[elements];  /* load element */
  if (x < min) { min = x; }
  if (x > max) { max = x; }
  jump minmaxhelp (elements+4, limit, min, max);
}

minmax(bits32 a) {
  jump minmaxhelp(a+4, a+4+4*bits32[a],
                  bits32[a+4], bits32[a+4]);
}
```

What C–– isn't

Not write-once, run-anywhere (Java, ANDF)



Not all things to all compilers (UNCOL)

What C–– is

C–– encapsulates compiler technology that is well understood, but difficult to implement

Goals:

- **Almost as easy as emitting C.**
- **Almost as good as a custom code generator**

What technology should be encapsulated?

Hide compiler algorithms, machine details

- Instruction selection, register allocation, instruction scheduling
- Standard scalar, array, and loop optimizations
- Tail-call optimizations and inlining

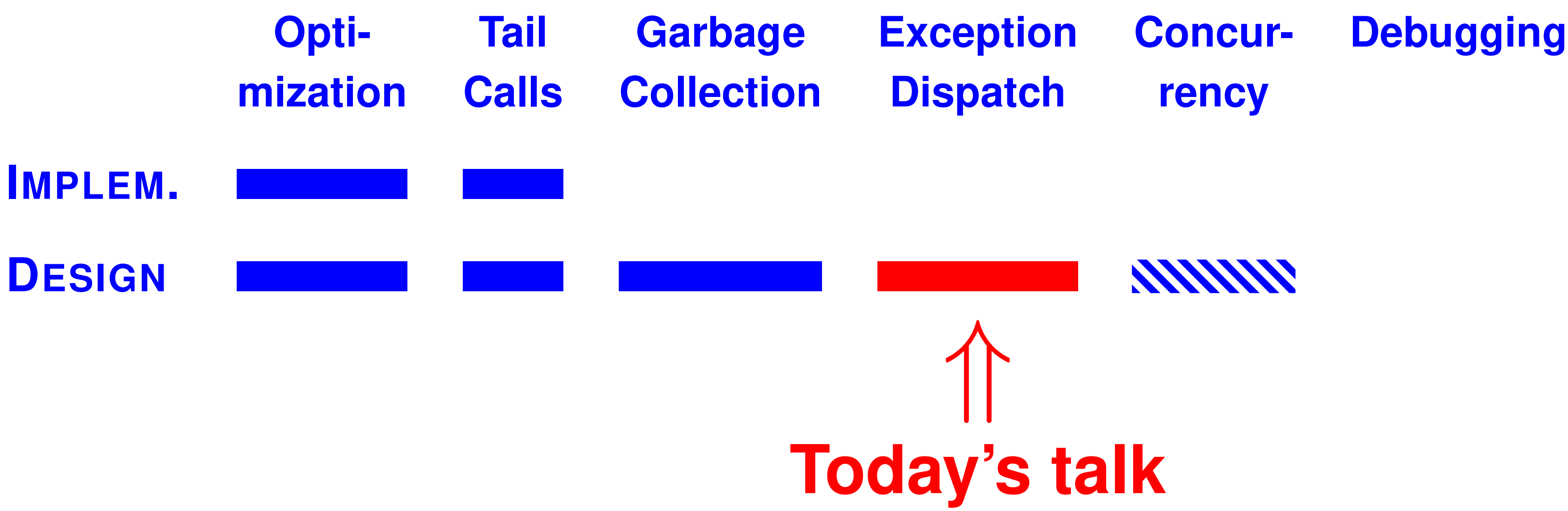
Encapsulate infrastructure for run-time services

- Garbage collection
- Exception dispatch
- Concurrency

If possible, support

- Source-level debugging

C–– status


Opti- mization
Tail Calls
Garbage Collection
Exception Dispatch
Concur- rency
Debugging
IMPLEM.
DESIGN
Today's talk

How to encapsulate?

Can't just implement garbage collector, exceptions

- One semantics doesn't fit all languages
- One cost tradeoff can't serve many clients

Instead, provide mechanisms or piece parts

- Different front ends can assemble in different ways
- Support standard implementations

Let the front end decide

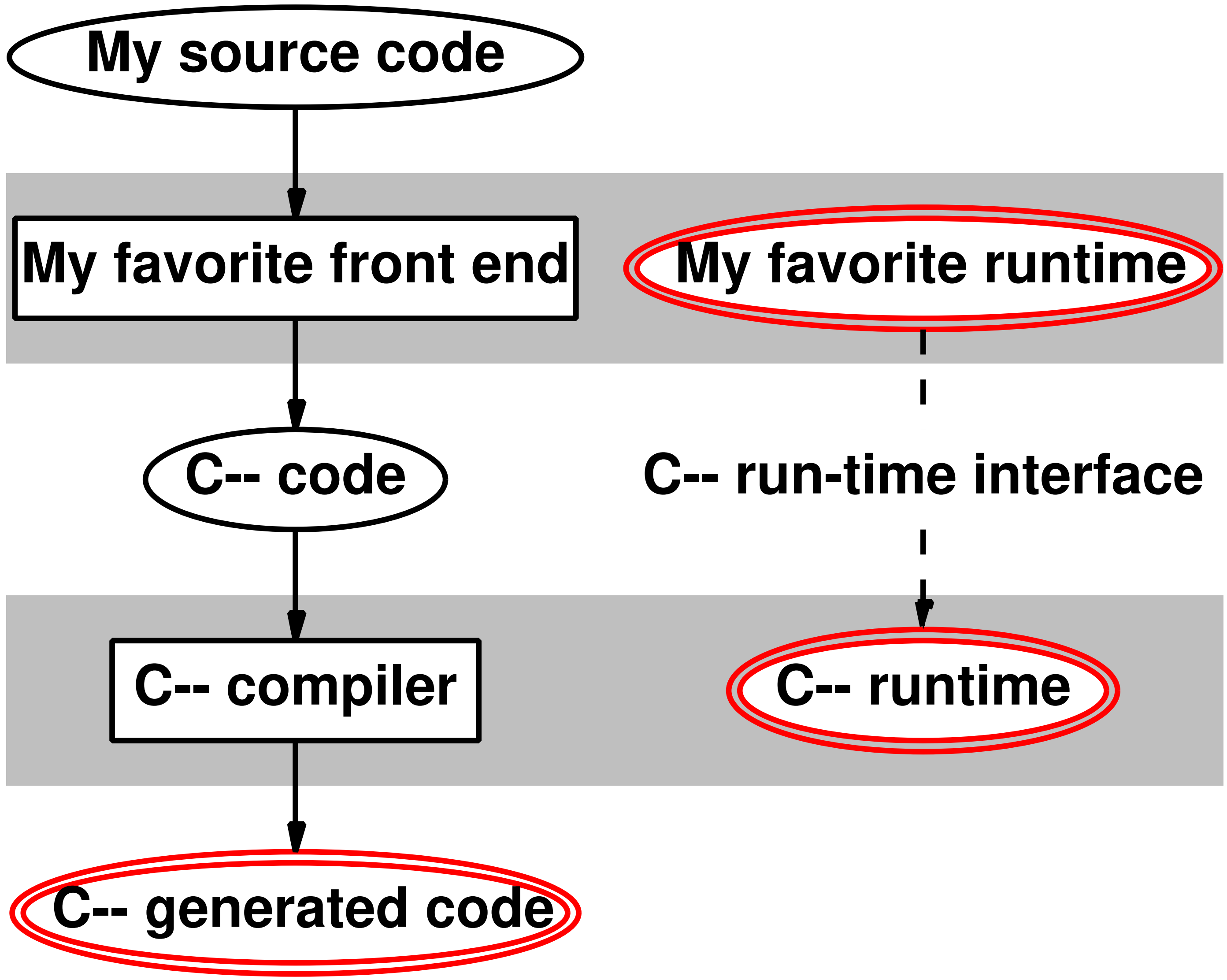
Exceptions

Contribution of this paper:

- **Identify how people do exceptions now**
- **Fit the choices into a single framework**
- **Precise, formal semantics—for optimization**

Step 1: run-time support

C–– in action (artist's conception)



Support for exceptions

Supports four implementation techniques:

- **Continuation-passing style (SML/NJ)**
- **Constant-time “stack-cutting” (OCaml)**
- **Unwind stack in run-time system (Poly M3)**
- **Custom code to unwind stack (Self)**

Hypothesis: supports multiple source languages

Cost model I: Stack cutting

Constant-time operation:

$$SP := \mathit{handler} \rightarrow SP;$$
$$\mathtt{goto}\ \mathit{handler} \rightarrow PC;$$

`cut to` *handler*

But

- **cannot restore callee-saves registers**
  (even in normal case, callee-saves less useful)
- *handler* **must be a C–– value**
- **typically imposes cost to enter scope of handler**

Cost model II: Stack unwinding

Visit each activation, frame by frame

- front-end runtime calls `NextActivation`
- C–– restores callee-saves registers as you go
- often used to find handler
- raising exception can be costly

But

- enter scope of handler with "zero overhead"
- can speed up by compiling unwinding code

Mechanism, Part I: run-time interface

Inspect, modify state of suspended computation

Abstract-data-type operations

- **stack walking:**
  `FirstActivation`, `NextActivation`
- **inspect and modify variables:** `FindLocalVar`, ...
- **unwind to activation:** `SetActivation`
- **change flow of control, parameters, results...**

Mechanism, Part II: continuations

C–– continuations

- **limited lifetime**
- **created on entry to a C–– procedure**
- **dies when its activation dies**

```
my_procedure (...) {
   ... main code (cuts or unwinds to k)...
   return answer;
 continuation k (exn_tag, arg):
   ... code for handlers ...
}
```

Using C–– continuations

Cutting:

- **language:** `cut to k (`*args*`)`
- **run-time interface:** `SetCutToCont`

Unwinding:

- **run-time interface: stack walk,** `SetUnwindCont`
- **language:**

  `return` <*continuation # / # of alternates*> `(`*results*`)`

  (Like CLU `signal`**, Smalltalk "nonlocal return")**
- **alternate returns can be very fast**

  (zero dynamic overhead in normal case)

Mechanism, Part III: annotations

Every call site may have multiple continuations

- **identified by annotations**
- **annotations inform the optimizer of altered control flow**
- **annotations trigger construction of tables**

Call syntax:

```
r = g( x ) also cuts    to k1
           also unwinds to k2, k3
           also returns to k4
           also aborts;
```

Unlikely to use all four in one front end!

Example

CLU tangent routine

```
tan = proc (x: real)
      returns (real)
      signals (ovflw, undflw);
  return (sin(x)/cos(x));
    resignal ovflw, undflw;
    except when zerodivide:
      signal ovflw;
    end;
end tan;
```

C–– translation

```
tan (bits64 x) {
  bits64 t1, t2;
  t1 = sin (x);
  t2 = cos (x);
  t1 = clu_fdiv (t1, t2)
       also returns
       to ov, un, zd;
    return <2/2> t1;

  continuation ov:
    return <0/2>;
  continuation un:
    return <1/2>;
  continuation zd:
    return <0/2>;
}
```

Mechanism, Part IV: spans

C–– spans

- **enclose any statements, procedures**
- **mark scope of handlers, also other info**
- **details in PPDP'99**
  (Peyton Jones, Ramsey, & Reig)

C–– covers the implementation space

Stack walk required?

Execute in	No	Yes
Generated code	`cut to`	`return` <m/n>
Run-time system	`SetCutToCont`	`SetActivation` `SetUnwindCont`

Where do we go from here?

Validate design experimentally

New compiler: Quick C––

- **Flexible compiler**
- **Multiple machines**
- **Fast compilation**
- **Easy to specialize (e.g., calling conventions)**
- **High-quality code not an aim (model is `lcc`)**

Plan for

1. **Clean implementation**
2. **Client front ends**
3. **Optimizers**

Conclusions

Combining language and runtime wins

Compiler-target language as design problem

- **encapsulate our well-understood technology**
- **avoid "heat death from special cases"**

Finally

C-- wants YOU!

`http://www.cminusminus.org`