The TclQuadcode Compiler

Status report on Tcl type analysis and code generation

Donal Fellows
orcid.org/0000-0002-9091-5938

Kevin Kenny

What is going on?

- Want to Make Tcl Faster
  - Everyone benefits
  - Lehenbauer Challenges
- 2 times faster ("Perl" territory)
  - Better algorithms
  - Better buffer management
  - Bytecode optimization
- 10 times faster ("C" territory)
  - Needs more radical approach

Generating Native Code is Hard

- Going to 10 times faster requires native code
  - Bytecode work simply won’t do it
- But Tcl is a very dynamic language
  - Even ignoring command renaming tricks
- Native code ***needs*** types
- Many platforms

Let’s Go to LLVM!

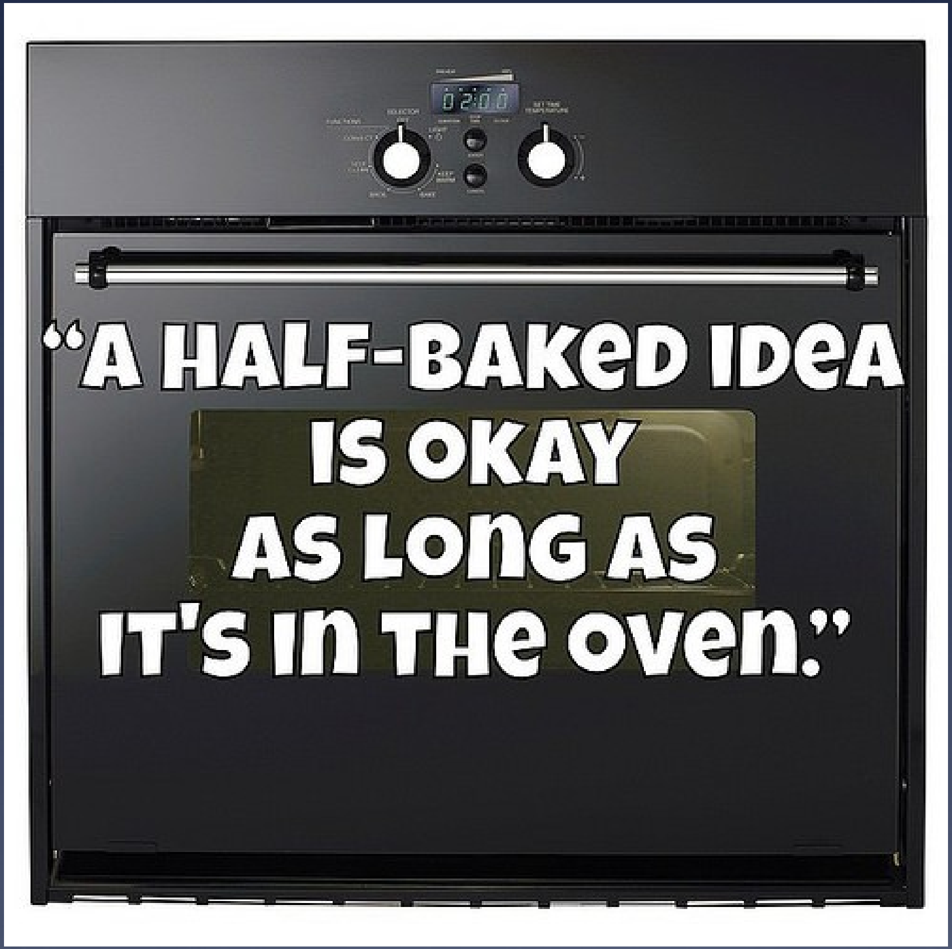
- Solves many problems
  - Optimization
  - Native code issuing
  - Runtime loading
- LLVM Intermediate Representation (IR)
  - Effectively a virtual assembly language target
- Existing Tcl package!
  - **llvmtcl** by Jos Decoster
- Introduces problems though
  - LLVM’s idea of “throw an error” is to panic with a gnostic error message

How to get to LLVM?

- Still need those pesky types
- Still need fixed semantics
- We need a *new bytecode*!

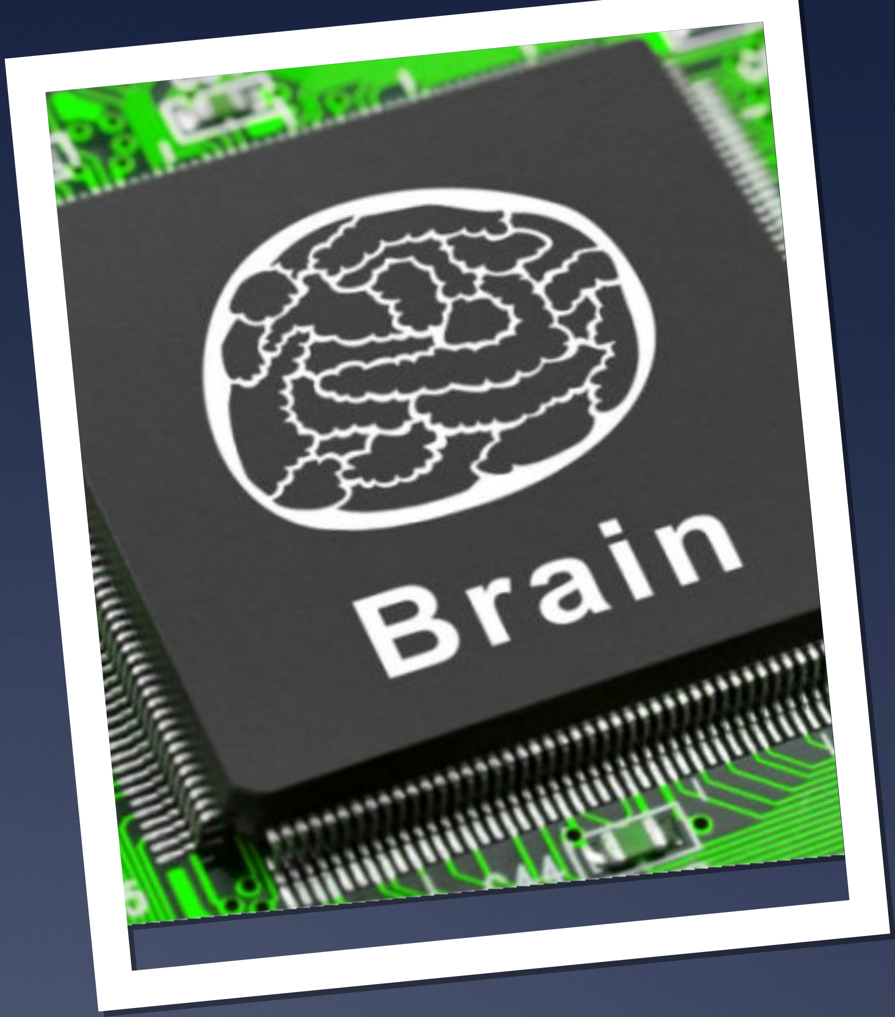
Quadcode

- Designed to help:
  - Simple translation from Tcl bytecode
  - More amenable to analysis


"A HALF-BAKED IDEA
IS OKAY
AS LONG AS
IT'S IN THE OVEN."

Tcl Analysis with

Quadcode



Quadcode

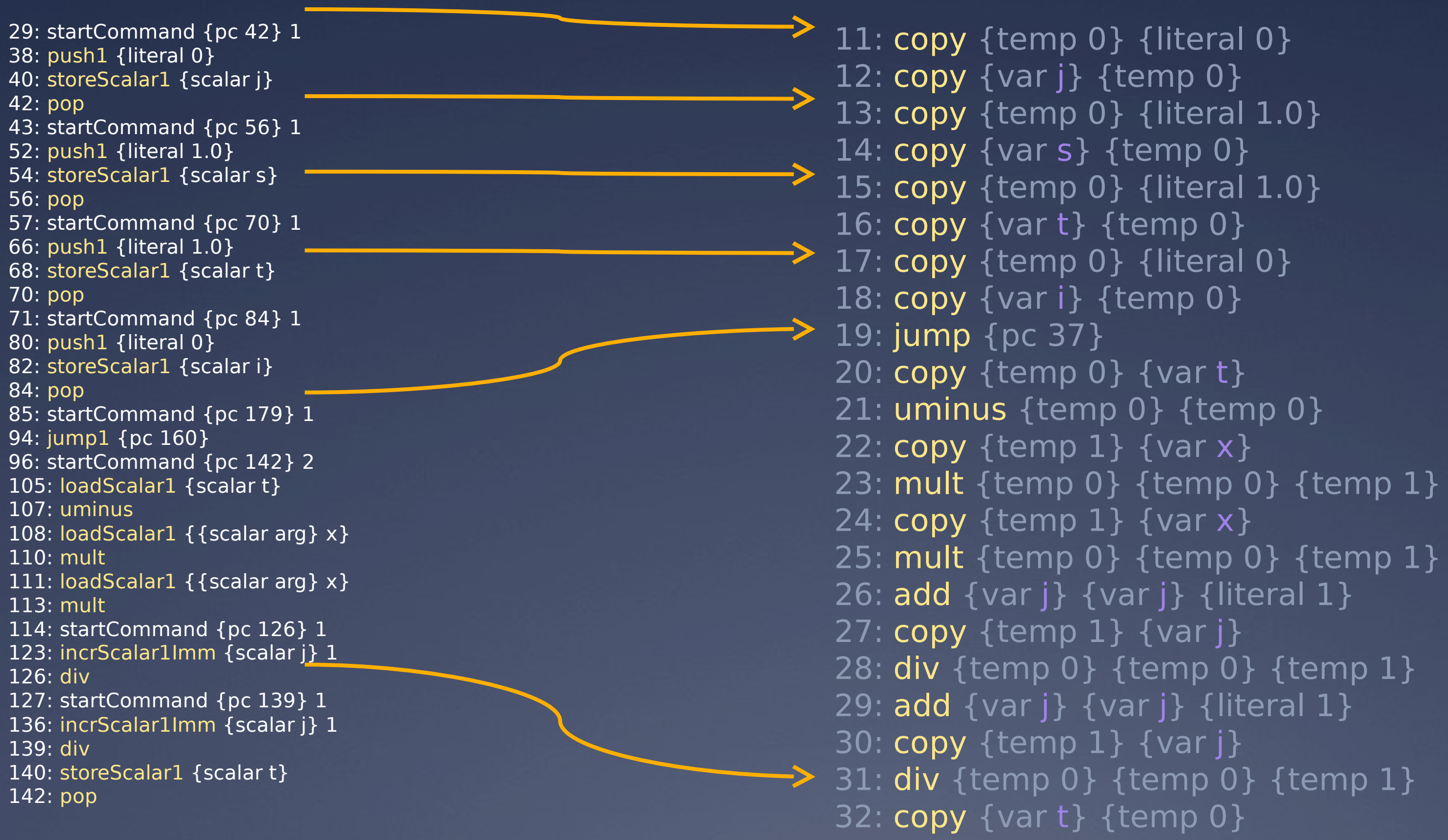
- Based on three-address code assembly
  - The Tcl code:
    `set a [expr { $b + 1 }]`
  - Equivalent Tcl bytecode:
    `loadScalar % b; push “1”; add; storeScalar % a`
  - Equivalent (optimized) quadcode:
    `add {var a} {var b} {literal 1}`
- No stack
  - Temporary variables used as required

Example: Tcl code to bytecode

```
proc cos {x {n 16}} {
  set x [expr {double($x)}]
  set n [expr {int($n)}]
  set j 0
  set s 1.0
  set t 1.0
  set i 0
  while {[incr i] < $n} {
     set t [expr {
        -$t*$x*$x / [incr j] / [incr j]
     }]
     set s [expr {$s + $t}]
  }
  return $s
}
```

```
    …
29: startCommand {pc 42} 1
38: push1 {literal 0}
40: storeScalar1 {scalar j}
42: pop
43: startCommand {pc 56} 1
52: push1 {literal 1.0}
54: storeScalar1 {scalar s}
56: pop
57: startCommand {pc 70} 1
66: push1 {literal 1.0}
68: storeScalar1 {scalar t}
70: pop
71: startCommand {pc 84} 1
80: push1 {literal 0}
82: storeScalar1 {scalar i}
84: pop
85: startCommand {pc 179} 1
94: jump1 {pc 160}
96: startCommand {pc 142} 2
105: loadScalar1 {scalar t}
107: uminus
108: loadScalar1 {{scalar arg} x}
110: mult
111: loadScalar1 {{scalar arg} x}
113: mult
114: startCommand {pc 126} 1
123: incrScalar1Imm {scalar j} 1
126: div
127: startCommand {pc 139} 1
136: incrScalar1Imm {scalar j} 1
139: div
140: storeScalar1 {scalar t}
142: pop
    …
```

Example: bytecode to quadcode



```
29: startCommand {pc 42} 1
38: push1 {literal 0}
40: storeScalar1 {scalar j}
42: pop
43: startCommand {pc 56} 1
52: push1 {literal 1.0}
54: storeScalar1 {scalar s}
56: pop
57: startCommand {pc 70} 1
66: push1 {literal 1.0}
68: storeScalar1 {scalar t}
70: pop
71: startCommand {pc 84} 1
80: push1 {literal 0}
82: storeScalar1 {scalar i}
84: pop
85: startCommand {pc 179} 1
94: jump1 {pc 160}
96: startCommand {pc 142} 2
105: loadScalar1 {scalar t}
107: uminus
108: loadScalar1 {{scalar arg} x}
110: mult
111: loadScalar1 {{scalar arg} x}
113: mult
114: startCommand {pc 126} 1
123: incrScalar1Imm {scalar j} 1
126: div
127: startCommand {pc 139} 1
136: incrScalar1Imm {scalar j} 1
139: div
140: storeScalar1 {scalar t}
142: pop
```

```
11: copy {temp 0} {literal 0}
12: copy {var j} {temp 0}
13: copy {temp 0} {literal 1.0}
14: copy {var s} {temp 0}
15: copy {temp 0} {literal 1.0}
16: copy {var t} {temp 0}
17: copy {temp 0} {literal 0}
18: copy {var i} {temp 0}
19: jump {pc 37}
20: copy {temp 0} {var t}
21: uminus {temp 0} {temp 0}
22: copy {temp 1} {var x}
23: mult {temp 0} {temp 0} {temp 1}
24: copy {temp 1} {var x}
25: mult {temp 0} {temp 0} {temp 1}
26: add {var j} {var j} {literal 1}
27: copy {temp 1} {var j}
28: div {temp 0} {temp 0} {temp 1}
29: add {var j} {var j} {literal 1}
30: copy {temp 1} {var j}
31: div {temp 0} {temp 0} {temp 1}
32: copy {var t} {temp 0}
```

Quadcode Analysis

- Code is converted to Static Single Assignment (SSA) form
  - Variables assigned only once
  - Phi (φ) instructions used to merge variables at convergences (after if-branches and in loops)
- Lifetime analysis
  - Corresponds to where to use `Tcl_DecrRefCount`
- Type analysis
  - What type of data actually goes in a variable?

Example: Tcl code to cleaned-up quadcode

```
proc cos {x {n 16}} {
  set x [expr {double($x)}]
  set n [expr {int($n)}]
  set j 0
  set s 1.0
  set t 1.0
  set i 0
  while {[incr i] < $n} {
     set t [expr {
        -$t*$x*$x / [incr j] / [incr j]
     }]
     set s [expr {$s + $t}]
  }
  return $s
}
```

```
0: param {var x} {arg 0}
1: param {var n} {arg 1}
2: invoke {var x} {literal tcl::mathfunc::double} {var x}
3: invoke {var n} {literal tcl::mathfunc::int} {var n}
4: copy {var j} {literal 0}
5: copy {var s} {literal 1.0}
6: copy {var t} {literal 1.0}
7: copy {var i} {literal 0}
8: jump {pc 18}
9: uminus {temp 0} {var t}
10: mult {temp 0} {temp 0} {var x}
11: mult {temp 0} {temp 0} {var x}
12: add {var j} {var j} {literal 1}
13: div {temp 0} {temp 0} {var j}
14: add {var j} {var j} {literal 1}
15: div {temp 0} {temp 0} {var j}
16: copy {var t} {temp 0}
17: add {var s} {var s} {temp 0}
18: add {var i} {var i} {literal 1}
19: lt {temp 0} {var i} {var n}
20: jumpTrue {pc 9} {temp 0}
21: return {} {var s}
```

Note that this is before SSA analysis

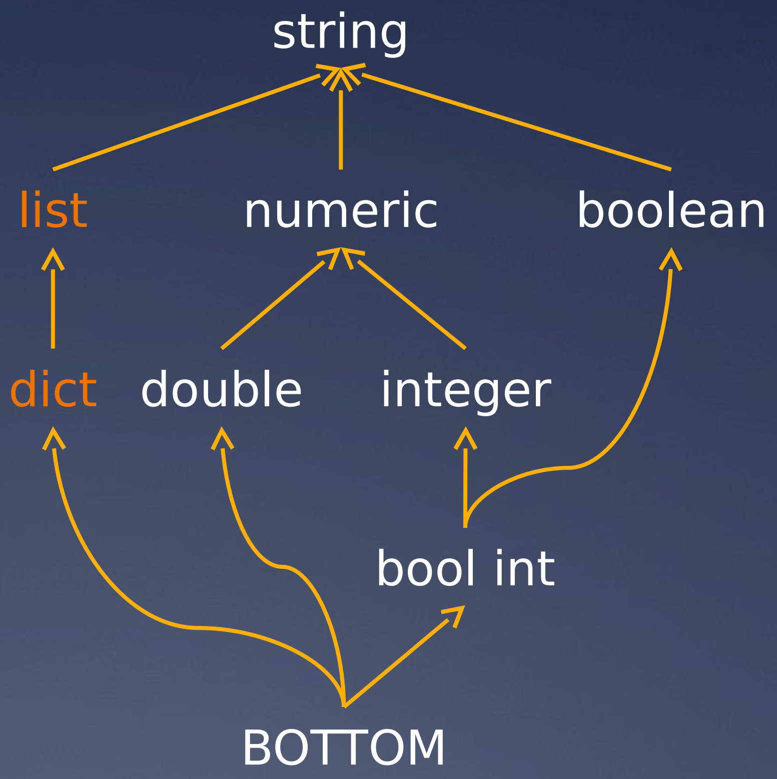
Example: In SSA form

```
0: param {var x 0} {arg 0}
1: param {var n 1} {arg 1}
2: invoke {var x 2} {literal tcl::mathfunc::double} {var x 0}
3: invoke {var n 3} {literal tcl::mathfunc::int} {var n 1}
4: copy {var j 4} {literal 0}
5: copy {var s 5} {literal 1.0}
6: copy {var t 6} {literal 1.0}
7: copy {var i 7} {literal 0}
8: jump {pc 18}
9: uminus {temp 0 9} {var t 21}
10: mult {temp 0 10} {temp 0 9} {var x 2}
11: mult {temp 0 11} {temp 0 10} {var x 2}
12: add {var j 12} {var j 19} {literal 1}
13: div {temp 0 13} {temp 0 11} {var j 12}
14: add {var j 14} {var j 12} {literal 1}
15: div {temp 0 15} {temp 0 13} {var j 14}
16: copy {var t 16} {temp 0 15}
17: add {var s 17} {var s 20} {temp 0 15}
18: confluence
19: phi {var j 19} {var j 4} {pc 8} {var j 14} {pc 17}
20: phi {var s 20} {var s 5} {pc 8} {var s 17} {pc 17}
21: phi {var t 21} {var t 6} {pc 8} {var t 16} {pc 17}
22: phi {var i 22} {var i 7} {pc 8} {var i 23} {pc 17}
23: add {var i 23} {var i 22} {literal 1}
24: lt {temp 0 24} {var i 23} {var n 3}
25: jumpTrue {pc 9} {temp 0 24}
26: return {} {var s 20}
```

INTRODUCED INSTRUCTIONS

The Types of Tcl

- Tcl isn’t entirely typeless
  - Our *values* have types
  - String, Integer, Double-precision float, Boolean, List, Dictionary, etc.
- But everything is a string
  - All other types are formally subtypes of string



Example: Determined Types

- Variable types inferred:
  - DOUBLE (i.e., proven to only ever contain a floating point number)
    - var x 0, var x 2,var t 8, var t 37, temp 0 16, ...
  - INT (i.e., proven to only ever contain an integer *of unknown width*)
    - var n 1, var n 4, var j 10, var i 12, var j 35, var j 22, var j 26, ...
  - INT BOOLEAN (i.e., proven to only ever contain the values 0 or 1)
    - var j 6, var i 9, temp 0 41, ...
- Return type inferred:
  - DOUBLE (i.e., always succeeds, always produces a floating point number)

Neat Tech along the Way

- Uses TclBDD as Reasoning Engine
  - Datalog is clean way to express complex programs
  - Good for computing properties
  - Stops us from going mad!
  - *(presented last year)*
- Might be possible to use quadcode itself as an bytecode-interpreted execution target
  - Totally not our aim, but it is quite a bit cleaner
  - *Not yet studied*

We’re at the Station…

Generating

LLVM IR

Generating LLVM

- LLVM Intermediate Representation (IR) is *very* concrete
  - Lower level than C
  - Virtual Assembler
- Each Tcl procedure goes to two functions
  1. Body of procedure
  2. “Thunk” to connect body to Tcl
- Each quadcode instruction goes to a non-branching sequence of IR opcodes
  - Keep pattern of basic blocks
  - Except branches, which branch of course

Compiling Instructions: Add

- Adding two floats is trivial conversion

```
%s = fadd double %phi_s, %tmp.08
```

- Adding two integers is not, as we don't know the bit width

- So call a helper function!

```
%j = call %INT @tcl.add(%INT %phi_j, %INT %k)
```

- The INT type is really a discriminated union

Compiling Instructions: Add

- Many ways to add
  - Which to use in particular situation?
- How we do it:
  - Look at the argument types (guaranteed known)
  - Look up TclOO method in code issuer to actually get how to issue code
    - Add the types to the method name
  - Unknown method handler generates normal typecasts
  - Just need to specify the “interesting” cases

Example: Issuing an Add

- Want to issue an add:

```
add {var a 1} {var b 2} {var c 3}
```

- Look up argument types:

```
{var b 2} → DOUBLE
{var c 3} → INT
```

- Call issuer method add(DOUBLE,INT)
  - Doesn’t exist, build from add(DOUBLE,DOUBLE) and typecaster

- End up with required instructions, perhaps:

```
%45 = call double @tcl.typecast.dbl(%INT %c.3)
%a.1 = fadd double %b.2 %45
```

Internal Standard Library Function

The Internal Standard Library

- Collection of Functions to be Inlined by LLVM Optimizer

- Implement many more complex operations

```
; casts from our structured INT to a double-precision float
define hidden double @tcl.typecast.dbl(%INT %x) #0 {
    ; extract the fields
    %x.flag = extractvalue %INT %x, 0
    %x.32 = extractvalue %INT %x, 1
    %x.64 = extractvalue %INT %x, 2

    ; determine what the 64-bit value is
    %is32bit = icmp eq i32 %x.flag, 0
    %value32bit = sext i32 %x.32 to i64
    %value = select i1 %is32bit, i64 %value32bit, i64 %x.64

    ; perform the cast and return it
    %casted = sitofp i64 %value to double
    ret double %casted
}
```

Optimization

- A critical step of IR generation is to run the optimizer
- Cleans up the code hugely
  - Inlines functions
  - Removes dead code paths
- We have much of Tcl API annotated to help the optimizer understand it
  - Documents guarantees and assumptions

Example: Optimized COS body

```
%6 = fmul double %phi_t64, %x
%7 = fmul double %6, %x
%tmp.04 = fsub double -0.000000e+00, %7

%8 = extractvalue %INT %phi_j62, 0
%9 = icmp eq i32 %8, 0
%10 = extractvalue %INT %phi_j62, 1
%11 = sext i32 %10 to i64
%12 = extractvalue %INT %phi_j62, 2
%x.6425.i43 = select i1 %9, i64 %11, i64 %12

%z.643.i44 = add i64 %x.6425.i43, 1
%cast = sitofp i64 %z.643.i44 to double
%tmp.05 = fdiv double %tmp.04, %cast
%z.643.i = add i64 %x.6425.i43, 2
%13 = insertvalue %INT { i32 1, i32 undef, i64 undef }, i64 %z.643.i, 2

%cast7 = sitofp i64 %z.643.i to double
%tmp.08 = fdiv double %tmp.05, %cast7
%s = fadd double %phi_s63, %tmp.08
```

The Other Types

- Lists and Dictionaries are treated as Strings
  - Mapped to a Tcl_Obj* reference
  - Lifetime management used to control reference counting efficiently
- Failing operations become tagged derived types
  - Failures cause jump to exception handling code
- The BOTTOM type only occurs in functions that cannot return
  - If they return, they do so by an error

Neat Tech along the Way

- Closures
  - Callbacks which capture local variables
- Locally-scoped Variables
  - Easy way to stop variables from one place spreading elsewhere
  - Prevented *many* nasty bugs

Example from Standard Library Builder

```
 # Create local LLVM function: tcl.int.32
 set f [$m local "tcl.int.32" int<-INT
ReadNone]
  params x
  build {
     my condBr [my isInt32 $x] $x32 $x64
  label x32:
     my ret [my int.32 $x]
  label x64:
     my ret [my cast(int) [my int.64 $x]]
  }

 # Make closure to create call to tcl.int.32
 my closure getInt32 {arg {resultName
""}} {
     my call [$f ref] [list $arg] $resultName
  }
```

Heading Out...

So, is this thing

FAST?

Performance Categories

- Numeric Code
  - Test pure integer functions with iterative fibonacci
  - Test floating point functions with cosines
- Reference-handling Code
  - Test string handling functions with complex string replacement
  - Test list handling functions with list joining
  - Test dictionary/array functions with counting words in a list
- Error-path Code
  - Test exception handling with non-trivial try usage

Performance

Category	Test	Time (µs)		Acceleration (%)	Target Reached?
		Uncompiled	Compiled		
Numeric	fib	12.15	0.4758	2453	✓✓
Numeric	cos	6.277	.3936	1495	✓✓
Reference	replac	[illegible]	[illegible]	40	✗
Reference	list	[illegible]	[illegible]	231	✓
Reference	wor	[illegible]	0	230	✓
Error	errortester	1	4.9	175	✓-ish

Looking Great!

Summary and Analysis

- Numeric code is *hugely* faster
  - Typically much more than 10 times faster!
- Reference management code is nicely faster
  - Often around 2–3 times faster
  - Automatically detecting how to unshare objects
  - String code largely unaffected
    - Critically dependent on buffer management
    - Might also be due to code used for testing
- Error code mostly faster
  - Could still do better, but not usually critical path

Going Fast!

Looking to the

Future

Where Next?

- Finish filling out translation from bytecode
  - Unset
  - Introspection
- Address slow speed of compilation
  - Resulting code is fast, but process to get to it…
- How to integrate into Tcl?
  - When to compile?
  - When to cache?
  - How to use LLVM practically?
  - What extensions to Tcl’s C API are needed?

Advanced Compilation

- Compilation between procedures
  - Can we use type info more extensively?
- Access to global variables
  - Currently local-variable only
  - Traces, variable scopes, etc.
- Other types of compileable things
  - Lambdas, methods, …

Longer-term Questions

- What changes should we do in Tcl in light of this?
  - Already some ideas:
    - Change incr to support floats
    - Some way to annotate suggested types on arguments
- If we bite the LLVM bullet, what other changes follow?
  - Need to link to C++ libraries to use LLVM
  - Implement official C++ API to Tcl?