Interpretation: The Final Frontier of Compilers

Martin Krastev, Chaos Group

Just an extra level of abstraction at runtime?

Compiled code

native code
processing uint (PU)

Interpreted code

foreign code
interpreter/VM
PU

Interpretation is key to optimizing compilers!

Optimizing compilers – machine-code generators that do (among other things) ahead-of-time interpretation so that runtime “interpretation” – a von-Neumann PU executing the machine version of the code, has less to do.

*‘The fastest code is code that is not executed (**at runtime**).’*

Over the next fourtish minutes

Ahead-of-time interpretation manifests vividly in partial evaluation (PE) – compile-time code evaluation and specialization. So let's do some PE from first principle! We will:

- Introduce a minimalistic 'calculator' language to quickly define ASTs we could tinker with
- Solve teething issues of inlining
- Devise a partial evaluator running on runtime properties of the code
- Write sample codes and run those through our tiny optimizer

For the sake of argument – TINL

This is Not Lisp (**TINL**, proncounced *tai-nul*) is a minimalistic LISP-like language, using familiar s-expressions with essential restrictions:

- no lists
- no array, vector, string or function types – just scalars
- no *defstruct* structured types either
- no *set/setq/setf* forms – variable initializations are effectively Single Static Assignments (SSA)
- no quoted expressions or macros
- no lambdas
- no NIL results – all expressions must return a value
- no bignum numeric type – only fixed-bitness integers
- no rational numeric type – only floating-point fractions
- no binary or octal literals – only decimal and hexidecimal (via *0x* prefix)
- no T/NIL predicate forms – *ifzero/ifneg expr then-expr else-expr* instead
- no *dotimes* et al loop forms – loops only via recursion

S-expressions for the uninitiated

S-expressons in TINL obey these syntactic principles:

- single-term expressions:

```
x                   // evaluate var x (not function x)
42                  // evaluate integral literal 42
```

- multi-term expressions – a sequence of terms separated by blanks and delimited by parentheses, where the ‘verb’ comes first – Polish notation:

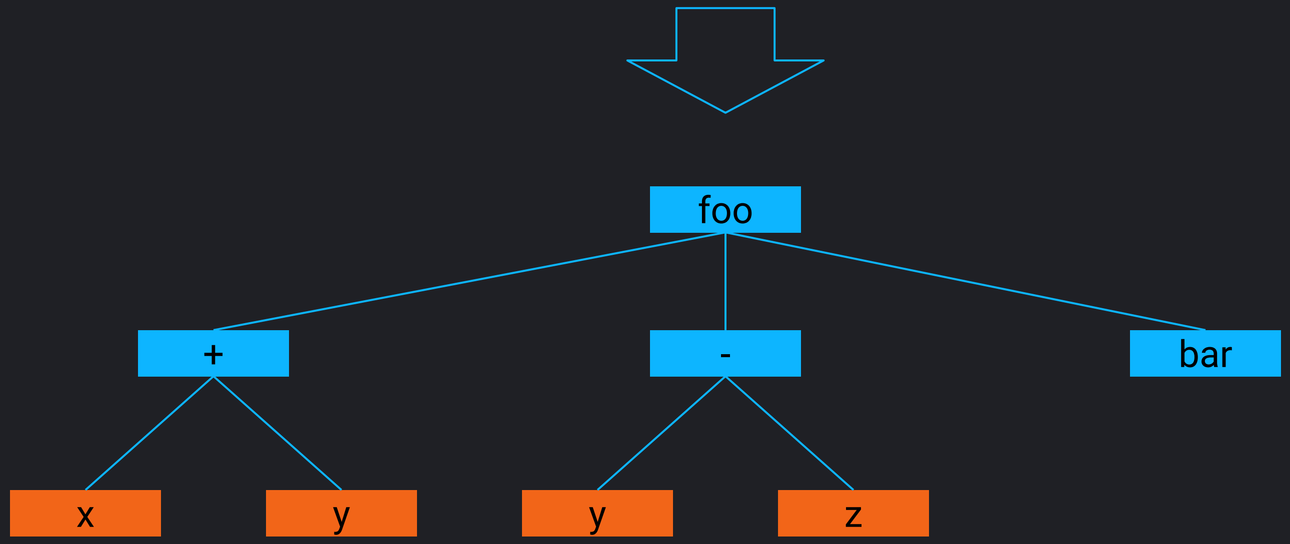
```
(foo x y z)         // invoke function foo on three arguments – vars x, y and z
(bar)               // invoke function bar of no arguments
```

A multi-term s-expression of multi-term s-expressions:

```
(foo (+ x y) (- y z) (bar))   // invoke foo on three args:   1. the sum of x and y
                              //                             2. the difference of y and z
                              //                             3. the result from bar
```

S-expressions – trees in disguise

(foo (+ x y) (- y z) (bar))



Input/output in TINL (similar to LISP)

Input:

- (readi32) – read an i32 from standard input
- (readf32) – read an f32 from standard input

Output:

- (print x) – print value of x to stardard output, return x
- return value of the root expression

Control flow in TINL (like in LISP)

Control flow follows these basic rules:

- explicit – function invocations – (foo)
- function arguments are evaluated in order of passing – (bar 1 2 3)
- ..unless invoking branching functions ifpred-then-else – (ifzero x (foo) (bar)) – either foo or bar
- let-expressions

Let-expressions in TINL (like in LISP)

Let-expressions – nested scopes that introduce symbols for reuse by sequentially-executed sub-expressions; last sub-expression is the result of the let-expression:

```
(let ((x 42) (y 43))      // initialize locally-scoped x and y to 42 and 43, respectively
  (print x)               // print x
  (print y)               // then print y
  (+ x y))                // return sum of x and y
```

Functions in TINL (like in LISP)

A defun statement defines a function – a named form of let-expressions, executed only by invocation.

```
(let ((x 42) (y 43))        // initialize locally-scoped x and y to 42 and 43, respectively
    (print x)               // print x
    (print y)               // then print y
    (+ x y))                // return sum of x and y
```

```
(defun foo(x y)             // define foo as function of x and y – no initialization!
    (print x)
    (print y)
    (+ x y))
(foo 42 43)                 // invoke foo for x = 42, y = 43
```

Function inlining in TINL

Inlining is the opposite to turning a let-expression into a function:

```
(defun bar(x y)
   (print (* x y)))
(bar 7 8)                    // invoke bar for x = 7, y = 8
```

```
(let ((x 7) (y 8))
   (print (* x y)))          // inline bar for x = 7, y = 8
```

Return-type evaluation (like in LISP)

The problem of return-type evaluation:

```
(defun foo() 42)                    // foo -> i32, via literal 42
(defun bar(x) x)                    // bar -> typeof(x)
(defun bas(x) (ifzero x 42 43.0))   // could be i32 or f32, via branching function ifzero-then-else
```

Deciding the return type of bas requires evaluating bas for x, ergo the problem of return-type evaluation == problem of general evaluation. The quirks of dynamism!

Code analysis – two sides to every story

Two distinct forms of code analysis. Expressed in loose human equivalents:

- Static analysis
  - A programmer staring at some code in the editor.
- Dynamic analysis
  - A programmer tracing that code in the debugger.

The realm of static analysis

Static analysis tries to tell things about a piece of code 'at a glance', without executing that piece of code. The realm of static analysis is static properties of the code.

```
(defun foo() 42)   // obviously foo -> i32 42
```

Once arguments are introduced:

```
(defun bar(x) x)   // obviously bar -> typeof(x)
```

Full observability of callees at call sites allows specialization for the given arguments:

```
(defun bar(x) x)
(bar 42)           // (bar 42) -> i32 42, by specializing bar for 42
```

The realm of dynamic analysis

Dynamic analysis tries to tell things about a piece of code at execution. The realm of dynamic analysis is control flow and computation results.

```
(defun foo(x y z) (+ x y z))
(foo 1 2 3)
```

Specializing for the call site and executing the computation, we conclude that (foo 1 2 3) -> i32 6.

Our use of both types of analysis

Today we will employ:

- minimal static analysis – limited type propagation at AST build time
- extensive dynamic analysis – partial evaluation via ahead-of-time AST interpretation and subsequent AST optimization

But first – inlining

Dynamic analysis is agnostic of inlining, but we still need inlining for one mundane reason:

```
(defun foo(x) (ifzero x 42 43))
(foo 0)            // arg x is zero, foo optimizes to literal 42
(foo 1)            // arg x is non-zero, foo optimizes to literal 43
```

To optimize foo, we potentially need as many ‘copies’ of foo as the number of call sites – code de-deduplication.

For the sake of simplicity, we will *always* inline during optimisation, so we can freely apply transformations on that function without worrying we’d break other call sites.

Inlining: a shadow of a problem

Original code

```
(let ((pi 3.14159265))
    (defun answer() pi)
    (let ((pi 42)) (answer)))
```

Inlined defun ‘answer’

```
(let ((pi 3.14159265))
    (defun answer() pi)
    (let ((pi 42)) (let () pi)))
```

So, what *is* the answer?

- Not inlined – 3.14159265
- Inlined – 42

Inlining: a shadow of a problem

Original code

```
(let ((pi 3.14159265))
   (defun answer() pi)
   (let ((pi 42)) (answer)))
```

Inlined defun ‘answer’

```
(let ((pi 3.14159265))
   (defun answer() pi)
   (let ((pi 42)) (let () pi)))
```

So, what *is* the answer?

- Not inlined – 3.14159265
- Inlined – 42

A: We cannot rely on the given identifiers at inlining due to unintended variable shadowing (problem A)

Inlining: a shadow of a problem, cont’d

Original code

```
(let ((pi 3.14159265))
    (defun answer() pi)
    (let ((pi 42)) (answer)))
```

Inlined defun ‘answer’

```
(let ((pi 3.14159265))
    (defun answer() pi)
    (let ((pi 42)) (let () pi)))
```

Fabricating unique identifiers at declaration and copying those at inlining avoids unintended shadowing.

```
(let ((pi [id:n] 3.14159265))
    (defun answer() pi [id:n])
    (let ((pi [id:m] 42)) (let () pi [id:n])))
```

Inlining: a shadow of a problem, cont’d

What about recursive code?

```
(defun foo(x [id:n] y [id:m])
  (foo y [id:m] (+ x [id:n] 1)))
```

Expanding one level of recursion via inlining

```
(defun foo(x [id:n] y [id:m])
  (let ((x [id:n] y [id:m]) (y [id:m] (+ x [id:n] 1)))
    (foo y [id:m] (+ x [id:n] 1))))
```

Inlining: a shadow of a problem, cont’d

What about recursive code?

```
(defun foo(x [id:n] y [id:m])
  (foo y [id:m] (+ x [id:n] 1)))
```

Expanding one level of recursion via inlining

```
(defun foo(x [id:n] y [id:m])
  (let ((x [id:n] y [id:m]) (y [id:m] (+ x [id:n] 1)))
    (foo y [id:m] (+ x [id:n] 1))))
```

Inlining: a shadow of a problem, cont'd

What about recursive code?

```
(defun foo(x [id:n] y [id:m])
  (foo y [id:m] (+ x [id:n] 1)))
```

Expanding one level of recursion via inlining

```
(defun foo(x [id:n] y [id:m])
  (let ((x [id:n] y [id:m]) (y [id:m] (+ x [id:n] 1)))
    (foo y [id:m] (+ x [id:n] 1))))
```

Merely copying fabricated IDs brings to picking the wrong shadows in init expressions (problem B)

Inlining: a shadow of a problem, cont'd

What about recursive code?

```
(defun foo(x [id:n] y [id:m])
  (foo y [id:m] (+ x [id:n] 1)))
```

Expanding one level of recursion via inlining

```
(defun foo(x [id:n] y [id:m])
  (let ((x [id:n] y [id:m]) (y [id:m] (+ x [id:n] 1)))
    (foo y [id:m] (+ x [id:n] 1))))
```

Merely copying fabricated IDs brings to picking the wrong shadows in init expressions.

```
(defun foo(x [id:n] y [id:m])
  (let ((x [id:n] y [id:m]) (y [id:m] (+ x [id:n] 1)))
    (foo y [id:m] (+ x [id:n] 1))))
```

Solution is simple – make declarations invisible for expressions in the same init section (blue rectangle on the left).

Bits of unobtainium

Our dynamic analysis has access to the return value of an expression, and we will utilize that for all our optimisations. A return value in TINL has the following natural attributes:

- Type (one of i32, f32)
- Value

We extend those with these additional attributes:

- flag *literal* – only literals have participated in the computation of the value
- flag *sidefx* – value has participated in a side effect
- flag *incoherent* – value has undergone non-deterministic branching returning different types

Bits of unobtainium: flag *literal*

Flag *literal* – only literals have participated in the computation of the value.

Examples:

```
(+ 1 2 3)                      // arithmetics over literals

(ifneg -1 3.14 (readf32))      // branching with a literal predicate that choses a literal branch

(defun foo(x) x)               // result from a pure function
(foo 42)                       // when passed a literal
```

Counterexamples:

```
(ifzero (readi32) -1 42)       // branch with a non-literal predicate – result not a literal
```

Bits of unobtainium: flag *sidefx*

Flag *sidefx* – value has participated in a side effect.

As variables in TINL are immutable, side effects can come only via the built-in function print. Examples:

```
(print 3.14)            // print a literal, return same literal

(print (readi32))       // print an i32 value read from input, return same value
```

Bits of unobtainium: flag *incoherent*

Flag *incoherent* – the value has undergone non-deterministic branching returning different types.

Examples:

(ifzero (readi32) 3.14 42)   // expression returns either an f32 or an i32, based on input

Counterexamples:

(ifzero (readi32) 3.14 42.0)   // expression returns f32, regardless of input – not incoherent

Bits of unobtainium: composition

If we present every value as a composite of some set of arguments A_0 through A_n , specific to that value, then:

- flag *literal* is an intersection of its compositing args – $A_0 literal \cap A_1 literal \cap \ldots A_n literal$
- flag *sidefx* is a union of its compositing args – $A_0 sidefx \cup A_1 sidefx \cup \ldots A_n sidefx$
- flag *incoherent* is a union of its compositing args – $A_0 incoherent \cup A_1 incoherent \cup \ldots A_n incoherent$

Bits of unobtainium: what are they worth?

Having those extra three attributes derived at every AST node along the path of PE allows us to specialize the AST accordingly:

- For subtrees that produce a literal but don't exert sidefx – collapse the subtree to a literal node.
- For branching whose predicate is a literal
  - if the predicate has no sidefx – shortcut the branching via an edge from the parent to the taken branch.
  - if the predicate has sidefx – turn the branching into a let-expression of two sub-expressions – the predicate and the taken branch.
- For nodes whose value is not incoherent – update the type of the node to the type of the value.

Please note, that by convention a value cannot be both literal and incoherent!

Our vehicle for today – TINL AST

A TINL AST is a TINL-correct syntax tree comprising of these node semantics:

- ASTNODE_LET – let-expression ***or*** defun-statement
- ASTNODE_INIT – statement introduces a named variable in a let-expression ***or*** defun-statement
- ASTNODE_EVAL_VAR – variable evaluation expression
- ASTNODE_EVAL_FUN – function evaluation expression (i.e. an invocation)
- ASTNODE_LITERAL – literal expression – either integral (i32) or floating-point (f32)

DEFUN does not have a dedicted node type. Instead we re-purpose a LET node into a DEFUN statement that is a nop for linear execution. We differentiate LET expressions from DEFUN statements by the fact the latter are named while the former are not.

AST examples in TINL

(+ 1 2 3)	(let ((x 42)) x)	(defun foo(x) x) (foo 42)

ASTNODE_EVAL_FUN: i32 +
  ASTNODE_LITERAL: i32 1
  ASTNODE_LITERAL: i32 2
  ASTNODE_LITERAL: i32 3
	ASTNODE_LET: i32
	  ASTNODE_INIT: i32 x
	    ASTNODE_LITERAL: i32 42
	  ASTNODE_EVAL_VAR: i32 x
		ASTNODE_LET: unknown foo
		  ASTNODE_INIT: unknown x
		  ASTNODE_EVAL_VAR: unknown x
		ASTNODE_EVAL_FUN: unknown foo
		  ASTNODE_LITERAL: i32 42

node semantics
What does it do?

eval type
What type is the result?

identifier or literal
How is it known, or what does it refer to?

Example codes: non-recursive flow

```
(defun abs(x)                 // abs: |x|
    (ifneg x (- 0 x) x))
(defun pow3(x)                // pow3: x^3
    (* x x x))
(defun foo(x y)               // foo: |x^3| * y
    (* (abs (pow3 x)) y))
(foo -3 (readi32))
```

AST of invocation post-PE

```
ASTNODE_LET: i32
    ASTNODE_INIT: i32 x
        ASTNODE_LITERAL: i32 -3
    ASTNODE_INIT: i32 y
        ASTNODE_EVAL_FUN: i32 readi32
    ASTNODE_EVAL_FUN: i32 *
        ASTNODE_LITERAL: i32 27
        ASTNODE_EVAL_VAR: i32 y
```

Example codes: recursive flow

```
(defun fac(n)                    // fac: n!
   (ifzero n 1 (* n (fac (- n 1)))))
(fac 12)
```

AST of invocation post-PE

ASTNODE_LITERAL: i32 479001600

Example codes: recursive flow

```
(defun fib(x y n)            // fib: the n-th fibonacci after x, y
  (ifzero n y (fib y (+ x y) (- n 1))))
(fib 1 1 44)                 // compute the 46th fibonacci
```

AST of invocation post-PE

ASTNODE_LITERAL: i32 1836311903

Example codes: recursive flow, sidefx

```
(defun fib(x y n)          // fib: the n-th fibonacci after x, y w/ print
    (print x)
    (ifzero n (print y) (fib y (+ x y) (- n 1))))
(fib 1 1 3)                // print the first 5 fibonaccis
```

AST of invocation post-PE

```
ASTNODE_LET: i32
  ASTNODE_INIT: i32 x
    ASTNODE_LITERAL: i32 1
  ASTNODE_INIT: i32 y
    ASTNODE_LITERAL: i32 1
  ASTNODE_INIT: i32 n
    ASTNODE_LITERAL: i32 3
  ASTNODE_EVAL_FUN: i32 print
    ASTNODE_LITERAL: i32 1
  ASTNODE_LET: i32
    ASTNODE_INIT: i32 x
      ASTNODE_LITERAL: i32 1
    ASTNODE_INIT: i32 y
      ASTNODE_LITERAL: i32 2
    ASTNODE_INIT: i32 n
      ASTNODE_LITERAL: i32 2
    ASTNODE_EVAL_FUN: i32 print
      ASTNODE_LITERAL: i32 1
```

```
ASTNODE_LET: i32
  ASTNODE_INIT: i32 x
    ASTNODE_LITERAL: i32 2
  ASTNODE_INIT: i32 y
    ASTNODE_LITERAL: i32 3
  ASTNODE_INIT: i32 n
    ASTNODE_LITERAL: i32 1
  ASTNODE_EVAL_FUN: i32 print
    ASTNODE_LITERAL: i32 2
  ASTNODE_LET: i32
    ASTNODE_INIT: i32 x
      ASTNODE_LITERAL: i32 3
    ASTNODE_INIT: i32 y
      ASTNODE_LITERAL: i32 5
    ASTNODE_INIT: i32 n
      ASTNODE_LITERAL: i32 0
    ASTNODE_EVAL_FUN: i32 print
      ASTNODE_LITERAL: i32 3
    ASTNODE_EVAL_FUN: i32 print
      ASTNODE_LITERAL: i32 5
```

Optimisations achieved

- Constant folding & constant propagation **across calls**
- Type propagation **across calls**
- Dead code elimination

All of the above are non-exhaustive – only for evaluated paths!

Know thy limits

Purely dynamic analysis fails at some problems trivial for static analysis.

What is the return type of the 'abs' invocation?

```
(defun abs(x)
    (ifneg x (- 0 x) x))
(abs (readi32))
```

Both branches of the non-deterministic ifneg utilize the 'unknown' x, so whichever path the PE takes, one x will be resolved to i32, while the other path will remain with an unresolved x, thus the overall return type of the invocation will be unresolved!

AST of defun 'abs'

```
ASTNODE_LET: unknown abs
    ASTNODE_INIT: unknown x
    ASTNODE_EVAL_FUN: unknown ifneg
        ASTNODE_EVAL_VAR: unknown x
        ASTNODE_EVAL_FUN: unknown -
            ASTNODE_LITERAL: i32 0
            ASTNODE_EVAL_VAR: unknown x
        ASTNODE_EVAL_VAR: unknown x
```

Notable encounters of PE

- Arithmetic/logical ops in #if (expression) – out-of-band PE
- Meta programming – e.g. C++ template specialization & subsequent optimisations
- Proper meta programming – e.g. macros in LISP
- Explicit compile-time execution – e.g. Zig ‘comptime’ decorator
- Deferred/just-in-time (JIT) compilation w/ specialization – e.g. LLVM MCJIT
- User-directed PE – e.g. AnyDSL compiler framework for computational kernels

Q & A

martin.krastev@chaosgroup.com

---

[1] Compile-time execution, Wikipedia
https://en.wikipedia.org/wiki/Compile_time_function_execution
[2] Fujita, *Partial evaluation with LLVM*
https://llvm.org/devmtg/2008-08-23/llvm_partial.pdf
[3] Jones, Gomard, Sestoft et al, *Partial Evaluation and Automatic Program Generation*
https://www.itu.dk/~sestoft/pebook/jonesgomardsestoft-a4.pdf
[4] *AnyDSL - A Partial Evaluation Framework for Programming High-Performance Libraries*
https://anydsl.github.io/
[5] Krastev, *TINL*
https://github.com/blu/tinl