TyCL v2.0 (alpha)

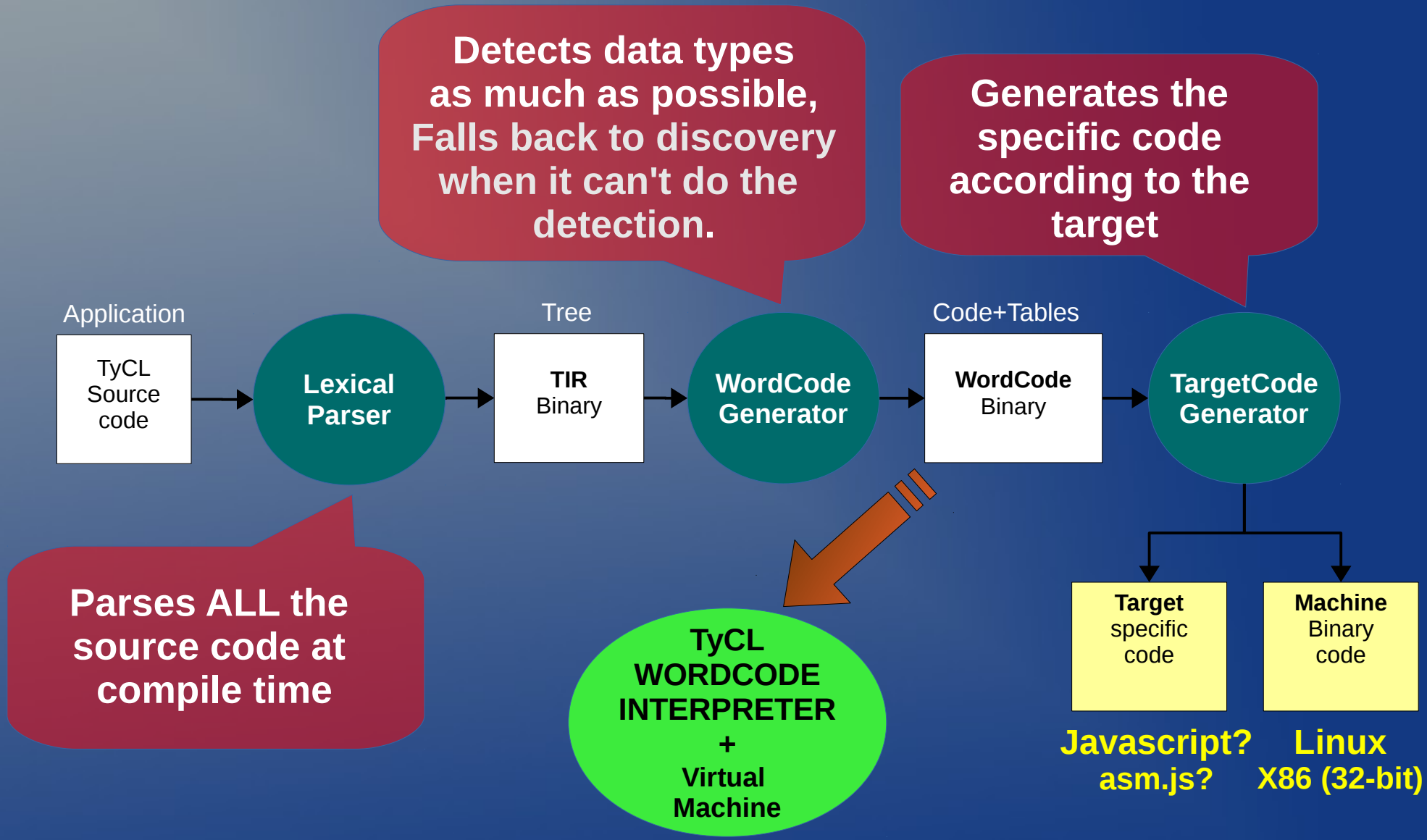
Typed **C**ommand **L**anguage

Andres Buss

What is TyCL?

- It is a flexible compiler capable of handling (a good part of) the Tcl/Tk syntax.
- It is a runtime-interpreter.
- It is the extended syntax of Tcl/Tk's language that the compiler understands. (Mostly the addition of “optionally” direct type declarations in the source code and the percent-commands)

Architecture



FORBIDS ANY DYNAMIC EVALUATION AT RUNTIME … no eval, no source commands

Features from TyCL 1.0

The “dynamic” parser

- Allows the creation/modification of syntax-rules

```
PARSER.addRule    "STATEMENT"  "any" \
        {";"  @SPACENL+~  @COMMENT~ @NATCMD @COMMAND}  “”  “”

PARSER.addRule “COMMAND”   “all”  \
        {@CMDNAME   @ARGUMENT*  @SPACE*~  @EOCMD}  \
        "COMMAND"    ""
```

- The percent commands (exectutes inside the compiler)

  %set, %proc, %include, %if, %eval, %puts, %exit

- Integrates the active-macro system

```
%macro FREE {o} "\[.MEM.free $o\]"

if {$v < 1} { set r $<FREE $p> }
```

Changes from TyCL 1.0

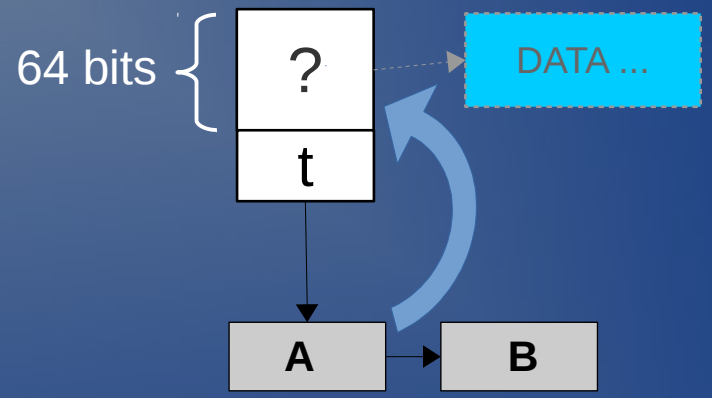
- The 'dot-notation' is no longer used to indicate sub-members of a variable, the Tcl's 'double-colon-notation' is used instead:

  Ex.: *set .g [foo.bar $a.x]* → *set ::g [foo::bar $a::x]*

- The 'dot-notation' was added to indicate children of a variable (specially for tk-widgets)

The new Type System

A TyCL type is a collection of functions/values (called descriptor) that handles the interaction of the raw-data that such type describes.



The type system:

- Allows the definition/creation of arbitrary types.
- Has an inheritance model.
- Access data by index, range, name, childName.

The new Type System

Three percent-commands where added

- *%typedef NEW_TYPE DESCRIPTOR*

```
%typedef myint {
    type integer
    ~size {u8:*} { return 1 }
    ~length {u8:*} { return 8 }
    ~names {} {}
    ~toString {str:*} { return [TOSTR_i8 $V] }
    ~indexGet {b8:* u8:index} { return b8:[expr $V & $index] }
    …
}
```

The new Type System

Three percent-commands where added

- *%metatype NAME PARAMETERS DESCRIPTOR*

```
%metatype cint {i32:max} {
    type i32
    ~set {cint:*} {
        if {$V > $max} { ERROR “Value out of boundary” }
        NEXT $V
    }

    …
}
```

The new Type System

Three percent-commands where added

- *%type* ***NEW_TYPE*** *METATYPE* *ARGUMENTS*

%type ***tinyint*** *cint* *25*

set a tinyint:8

The WordCode Generator

Takes the Parser's AST and performs two stages manipulating tokens:

1. Identify and reduce.
   - Detects and keeps track of variables and their types
   - Performs any operations that can be resolved directly. Ex. Eliminate code dependant of a false condition.
   - Inline native and type-descriptor functions.

2. Transform
   - Transform tokens into word-codes

The WordCode Generator

```
set a 44
set b $a
set c [foo]

if {$b > $c} {
        puts "$b is greater than $c"
}
```

The WordCode Generator

```
set a 44
set b $a
set c [foo]

if {$b > $c} {
        puts "$b is greater than $c"
}
```

```
<SET> ()
   <VARPATH> ()
      <STRING> (str) 'a'
   <LIT> (i8) '44'
<SET> ()
   <VARPATH> ()
      <STRING> (str) 'b'
   <GETVAL> ()
      <VARPATH> ()
         <STRING> (str) 'a'
<SET> ()
   <VARPATH> ()
      <STRING> (str) 'c'
   <SUBCMD> ()
      <COMMAND> ()
         <VARPATH> ()
            <STRING> (str) 'foo'
<IF> ()
   <EXPR> ()
      <EOP> () '>'
         <GETVAL> ()
            <VARPATH> ()
               <STRING> (str) 'b'
         <GETVAL> ()
            <VARPATH> ()
               <STRING> (str) 'c'
```

The WordCode Generator

```
set a 44
set b $a
set c [foo]

if {$b > $c} {
        puts "$b is greater than $c"
}
```

```
<SET> ()
    <VAR> () 'c'
    <CALL> ()
        <VAR> () 'foo'
<IF> ()
    <GT> (b8)
        <LIT> (i8) '44'
        <VAR> () 'c'
    <BLOCK> ()
        <PUTS> ()
            <CONCAT> (str)
                <LIT> (str) '44 is greater than '
                <TOSTR> (str)
                    <VAR> () 'c'
```

```
<SET> ()
    <VARPATH> ()
        <STRING> (str) 'a'
    <LIT> (i8) '44'
<SET> ()
    <VARPATH> ()
        <STRING> (str) 'b'
    <GETVAL> ()
        <VARPATH> ()
            <STRING> (str) 'a'
<SET> ()
    <VARPATH> ()
        <STRING> (str) 'c'
    <SUBCMD> ()
        <COMMAND> ()
            <VARPATH> ()
                <STRING> (str) 'foo'
<IF> ()
    <EXPR> ()
        <EOP> () '>'
            <GETVAL> ()
                <VARPATH> ()
                    <STRING> (str) 'b'
            <GETVAL> ()
                <VARPATH> ()
                    <STRING> (str) 'c'
```

Extending the language/compiler

- Create/modify the syntax.
- Create arbitrary word-code tokens and reductions.
- Create arbitrary word-code transformations.
- Eventually: create Machine-Code generators

Extending the language

- Create new arbitrary types (full/derived/meta types). Including function-types.
- Create conversions (casting) between types.

  %cast i8 b8 { return [CAST_i8_b8 $V] }
- Create expr-operators

  %expr_operator "+" {i8:L i8:R} { return [ADD_i8 $L $R] }
- Create expr-functions

  %expr_function "cos" {f64:V} { return [COS_f64 $V] }

A simple performance analisys

The primes program: find the number of prime numbers between 0 and some number (n)

```
proc func {n} {
        set tot 0
        for {set i 1} {$i < $n} {incr i 1} {
                set flg 1
                for {set j 2} {$j < $i} {incr j 1} {
                        set r [expr $i % $j]                ;#set r [expr {$i % $j}
                        if {$r == 0} { set flg 0 }
                }

                if {$flg == 1} { incr tot 1 }
        }

        puts "number of primes: $tot"
}

func 10000
```

A simple performance analisys

The primes program: find the number of prime numbers between 0 and some number (n)

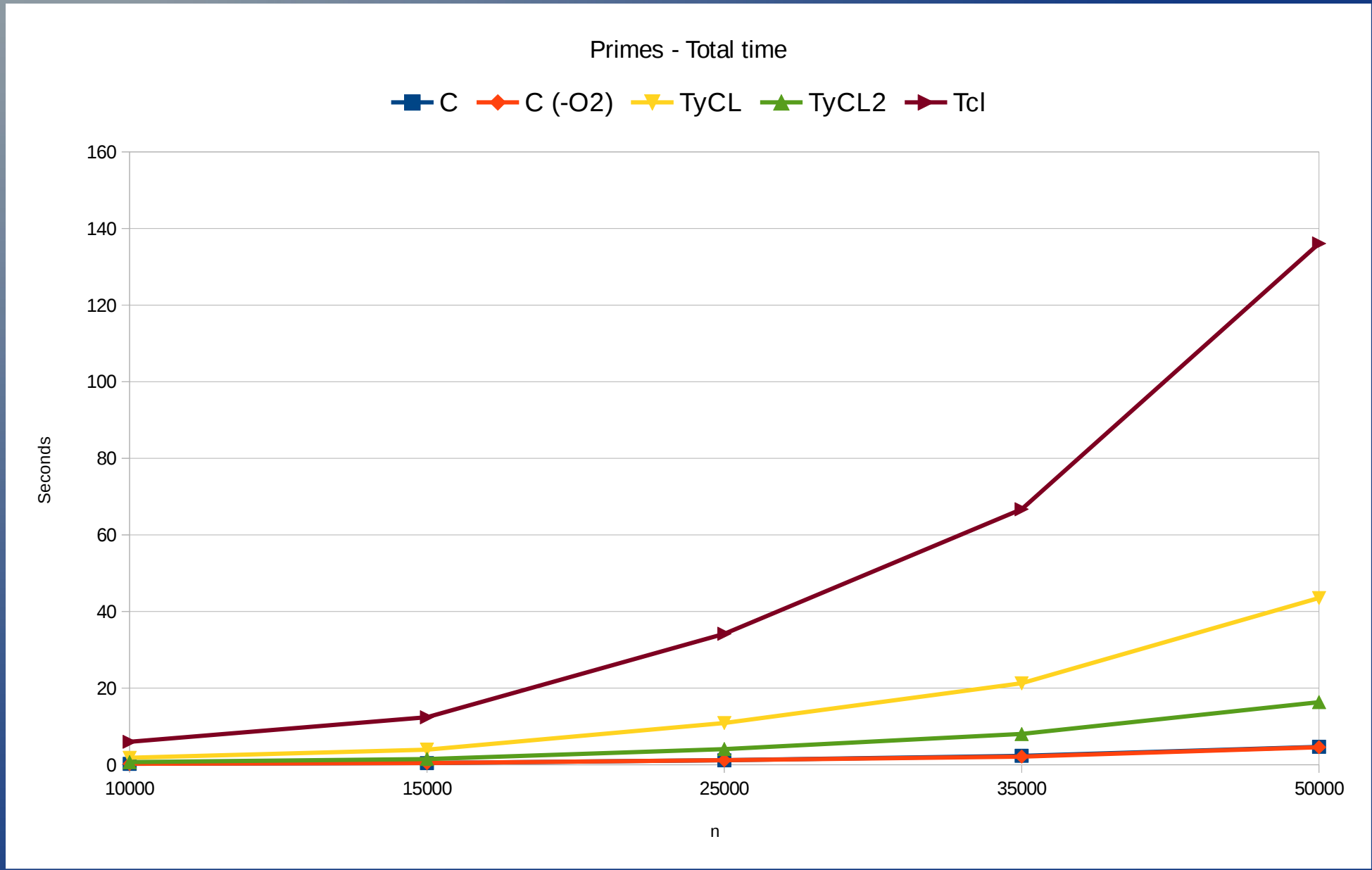
```
void func(int n) {
        int tot = 0;
        int i,j,r,flg;

        for(i=1; i<n; i++) {
                flg = 1;
                for(j=2; j<i; j++) {
                        r = i%j;
                        if(r == 0) { flg = 0; }
                }
                if(flg == 1) { tot += 1; }
        }

        printf("number of primes: %d\n",tot);
}

int main(void) {
        func(10000);
}
```

A simple performance analisys



Roadmap for v2.0 (final)

- ~~Have some “infrastructure” stabilization~~
- Add support for X86_64 (New assembler)
- Add official support for Javascript (asm.js)
- Have some documentation and a WEB page
- Release de source code (BSD licence)

More information:

Andres Buss

aabuss@otlettech.com

aabuss@gmail.com