


# Building a faster expression evaluator for LLDB

Andy Yankovsky, Google  @werat  
LLVM Developers' Meeting 2021

# Stadia for Visual Studio debugger

- Visual Studio extension implementing typical developer workflows
  - <https://github.com/googlestadia/vsi-lldb>
- Build & Run & Debug the game in the Cloud™
- Debugger is based on LLDB...
- ... and supports [NatVis!](#)

# Custom object visualizers (NatVis) #1

The image shows a Visual Studio IDE with two windows open: `Dummy.cpp` and `NatvisFile.natvis`.

**Dummy.cpp:**

```
1 #include <iostream>
2
3
4 struct Rectangle {
5     int height;
6     int width;
7 };
8
9 int main() {
10     Rectangle r1 = {10, 10};
11     Rectangle r2 = {5, 10};
12 }
```

**NatvisFile.natvis:**

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">
3     <Type Name="Rectangle">
4         <DisplayString Condition="height == width">A square with sides of {height}</DisplayString>
5         <DisplayString>A rectangle with height = {height} and width = {width}</DisplayString>
6         <Expand HideRawView="true">
7             <Item Name="height">height</Item>
8             <Item Name="width">width</Item>
9             <Item Name="area">height * width</Item>
10        </Expand>
11    </Type>
12 </AutoVisualizer>
```

A red arrow points to the `height * width` expression in the `area` item of the NatVis configuration.

**Locals Window:**

Name	Value	Type
└─ r1	A square with sides of 10	Rectangle
└─ height	10	int
└─ width	10	int
└─ area	100	int
└─ r2	A rectangle with height = 5 and width = 10	Rectangle
└─ height	5	int
└─ width	10	int
└─ area	50	int

# Custom object visualizers (NatVis) #2

The image shows a Visual Studio IDE with two windows open: `Dummy.cpp` and `NatvisFile.natvis`. The `Dummy.cpp` window shows a C++ program with a `Rectangle` struct and a `main` function that creates two rectangles: `r1` (width 10, height 10) and `r2` (width 5, height 10). The `NatvisFile.natvis` window shows an XML configuration for an `AutoVisualizer` for the `Rectangle` type. The configuration includes a `DisplayString` condition for squares and a list of `Item` elements for `height`, `width`, and `area`. Red boxes highlight the expressions used in the NatVis configuration: `height == width`, `height`, `width`, and `height * width`. A red note states: `~5 expressions for rendering a single object!`

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">
3   <Type Name="Rectangle">
4     <DisplayString Condition="height == width">A square with sides of {height}</DisplayString>
5     <DisplayString>A rectangle with height = {height} and width = {width}</DisplayString>
6     <Expand HideRawView="true">
7       <Item Name="height">{height}</Item>
8       <Item Name="width">{width}</Item>
9       <Item Name="area">{height * width}</Item>
10    </Expand>
11  </Type>
12 </AutoVisualizer>
```

Locals

Name	Value	Type
└─ r1	A square with sides of 10	Rectangle
└─ height	10	int
└─ width	10	int
└─ area	100	int
└─ r2	A rectangle with height = 5 and width = 10	Rectangle
└─ height	5	int
└─ width	10	int
└─ area	50	int

# Custom object visualizers (NatVis) #3

[clang/utils/ClangVisualizers/clang.natvis](#)

The whole visualizer is 84 lines >\_<

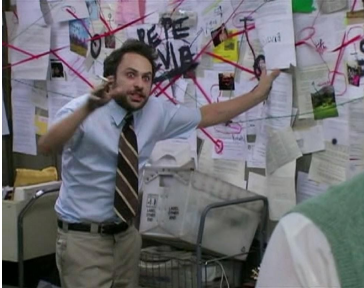
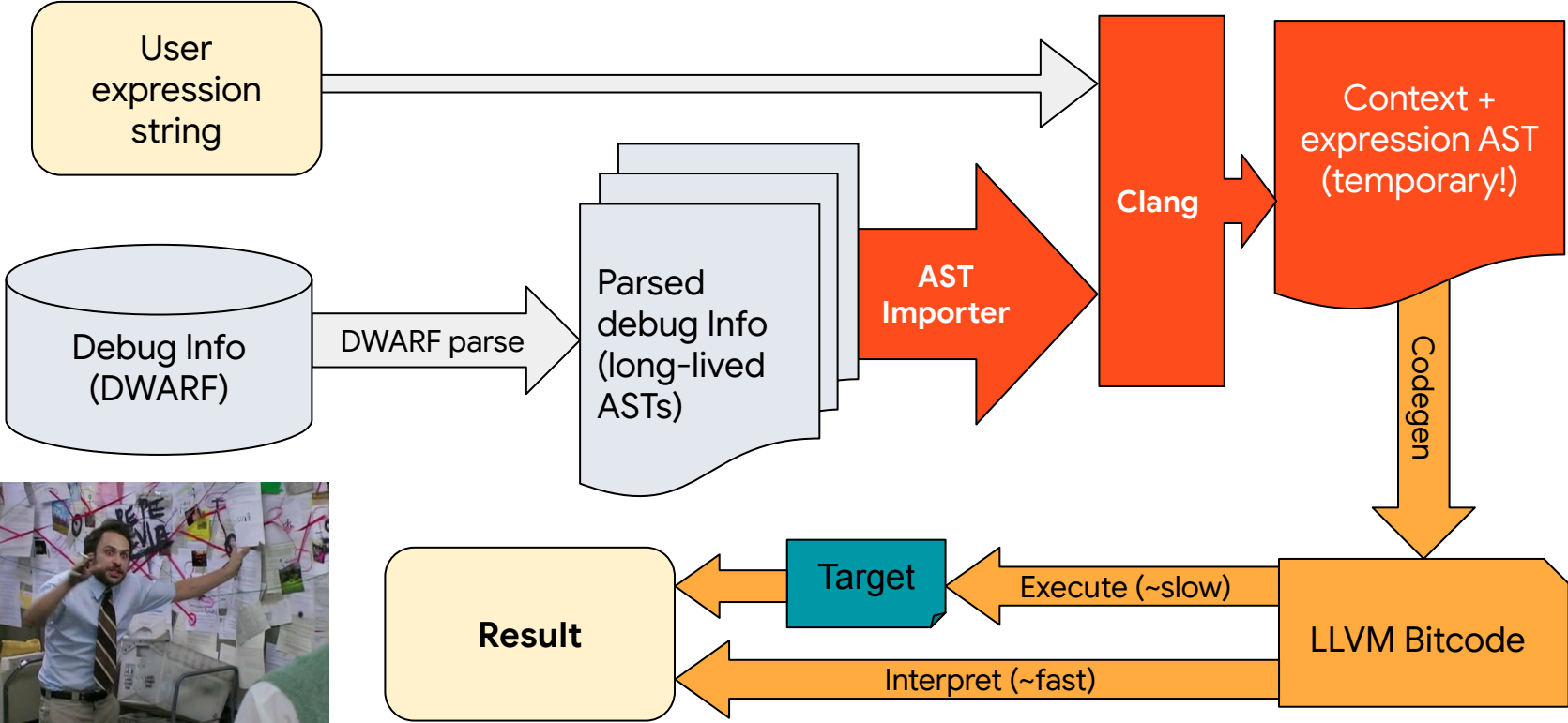
```
11 <Type Name="clang::Type">
12   <!-- To visualize clang::Types, we need to look at TypeBits.TC to determine the actual
13        type subclass and manually dispatch accordingly (Visual Studio can't identify the real type
14        because clang::Type has no virtual members hence no RTTI).
15
16        Views:
17        "cmn": Visualization that is common to all clang::Type subclasses
18        "poly": Visualization that is specific to the actual clang::Type subclass. The subtype-specific
19                <DisplayString> is typically as C++-like as possible (like in dump()) with <Expand>
20                containing all the gory details.
21        "cpp": Only occasionally used when we need to distinguish between an ordinary view and a C++-like view.
22   -->
23   <DisplayString IncludeView="cmn" Condition="TypeBits.TC==clang::LocInfoType::LocInfoType">LocInfoType</DisplayString>
24   <DisplayString IncludeView="cmn">{(clang::Type::TypeClass)TypeBits.TC, en}Type</DisplayString>
25   <!-- Dispatch to visualizers for the actual Type subclass -->
26   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::Builtin" IncludeView="poly">*(clang::BuiltinType *)this</DisplayString>
27   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::Pointer" IncludeView="poly">*(clang::PointerType *)this</DisplayString>
28   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::LValueReference" IncludeView="poly">*(clang::LValueReferenceType *)this</DisplayString>
29   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::RValueReference" IncludeView="poly">*(clang::RValueReferenceType *)this</DisplayString>
30   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::ConstantArray" IncludeView="poly">{(clang::ConstantArrayType *)this,na}</DisplayString>
31   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::ConstantArray" IncludeView="left">{(clang::ConstantArrayType *)this,view(left)na}</DisplayString>
32   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::ConstantArray" IncludeView="right">{(clang::ConstantArrayType *)this,view(right)na}</DisplayString>
33   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::IncompleteArray" IncludeView="poly">{(clang::IncompleteArrayType *)this,na}</DisplayString>
34   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::IncompleteArray" IncludeView="left">{(clang::IncompleteArrayType *)this,view(left)na}</DisplayString>
35   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::IncompleteArray" IncludeView="right">{(clang::IncompleteArrayType *)this,view(right)na}</DisplayString>
36   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::Typedef" IncludeView="poly">{(clang::TypedefType *)this,na}</DisplayString>
37   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::Typedef" IncludeView="cpp">{(clang::TypedefType *)this,view(cpp)na}</DisplayString>
38   <DisplayString Condition="TypeBits.TC==clang::Type::TypeClass::Attributed" IncludeView="poly">*(clang::AttributedType *)this</DisplayString>
```

# Expression evaluation in LLDB

- `lldb::SBValue::GetValueForExpressionPath()`
  - Expands nested expressions like `a->b[0].c[1]->d`
  - Very fast – ~0.1 ms per expression
- `lldb::SBFrame::EvaluateExpression()` // also exists for `SBTarget` and `SBValue`
  - Can handle almost any valid C++
  - Can be quite slow – ~50-100 ms per expression

... Can we make it faster?

# Expression evaluation in LLDB



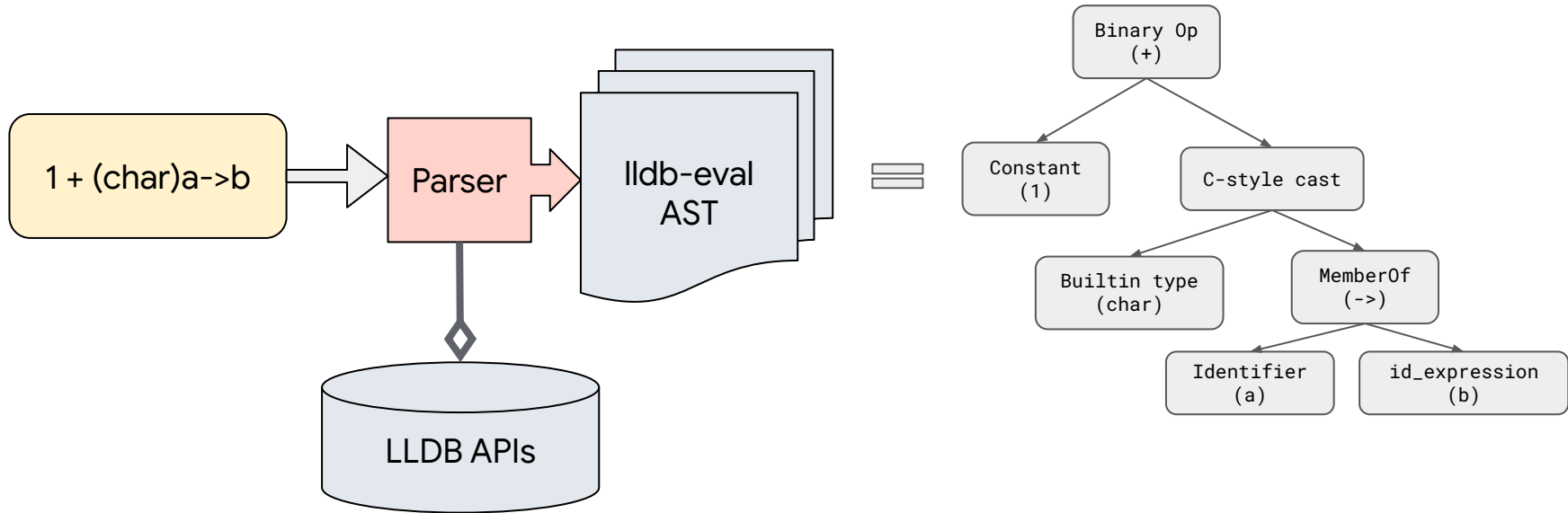
# We need to go faster – lldb-eval

- A library, (almost) drop-in replacement
  - <http://github.com/google/lldb-eval>
- Features:
  - Fast – <1 ms per evaluation
  - All basic operations – arithmetic, member access, type casts, etc.
  - Builtin intrinsic functions (e.g. `__log2`, `__findnonnull`)
- Limitations:
  - No user function calls (yet)
  - No constant-time evaluation (e.g. `Foo<1+2>::bar`)

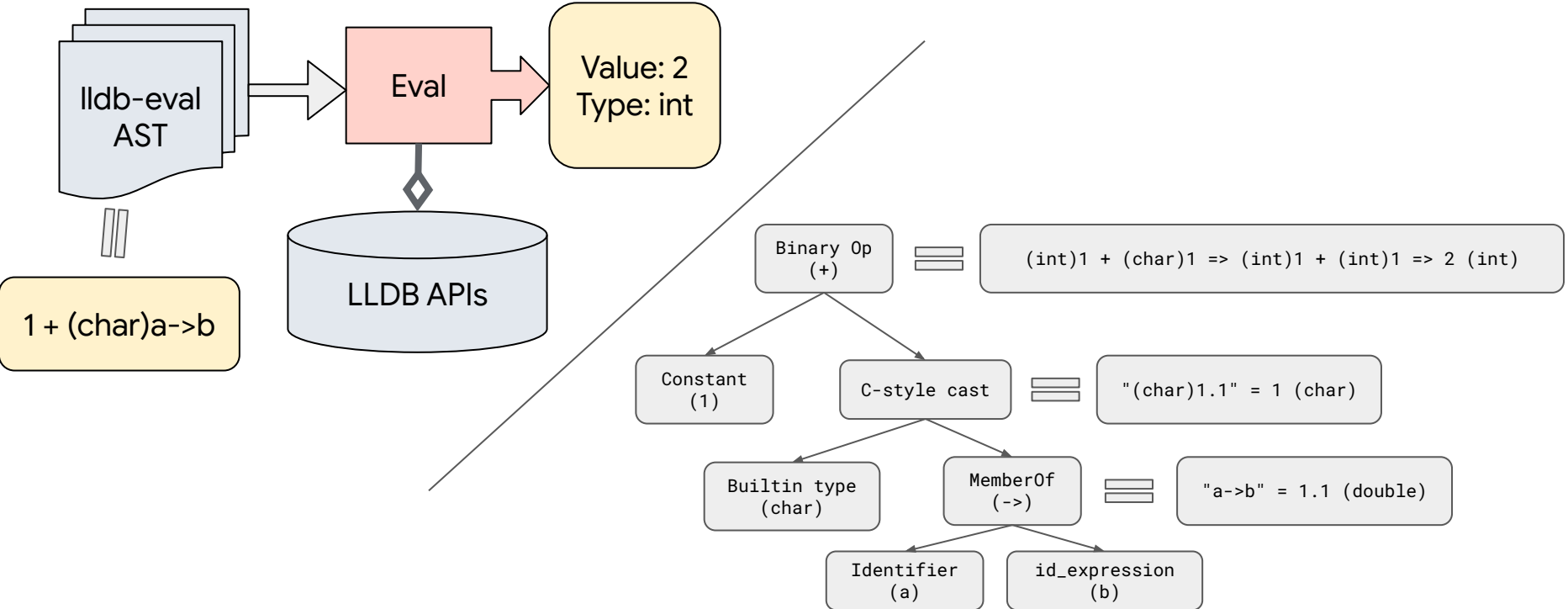




# Life of an expression in lldb-eval #1



# Life of an expression in lldb-eval #2



# How come lldb-eval is faster?

- LLDB uses Clang and Clang is a *compiler* – it needs to resolve everything and the expression AST may end up very large
- lldb-eval tries to be as lazy as possible – requests only the information needed
- lldb-eval is basically a re-implementation of Clang Frontend – specialized, hacky, fast!

## lldb-eval



Is it int? Looks like void\* to me

Let's iterate over this array and two others next to it in memory

I can dereference whatever I want

## C++ compiler



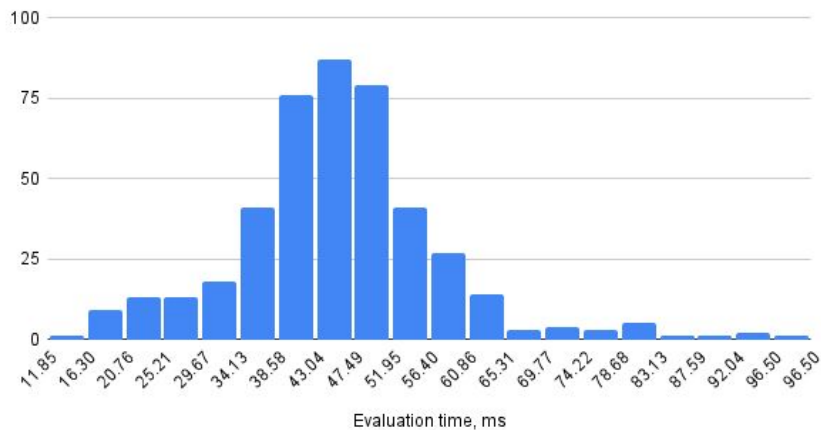
Ppp.. please.. i-initializ a reference first.. what? no, u can't change the t-t-target, sorry..

# Performance comparison – benchmark setup

- Unreal Engine 4, Infiltrator Demo
  - <https://www.youtube.com/watch?v=dO2rM-L-vdQ>
- Linux executable
  - Binary – ~150 MB
  - Symbols – ~1800 MB
- Expand an object of type UWorld
  - Triggers evaluation of ~300 expressions

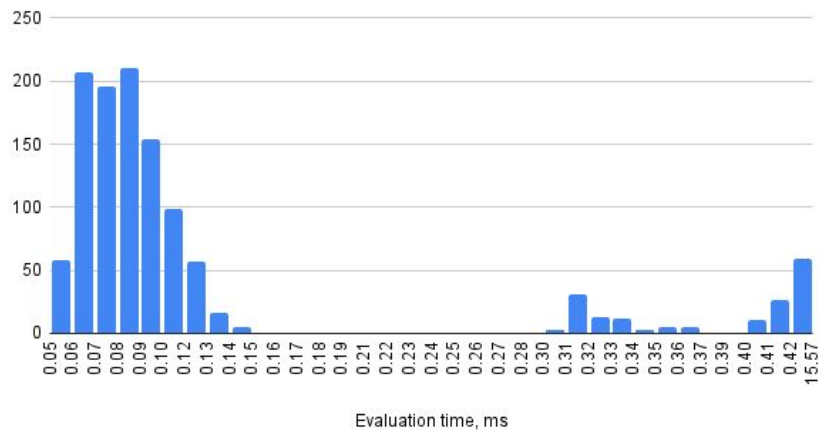
# Performance comparison

Evaluation time distribution, LLDB



300 expressions \* 50ms  $\approx$  **15 seconds!**

Evaluation time distribution, lldb-eval



300 expressions \* 0.3ms  $\approx$  **90 milliseconds!**

# References

- Blazing fast expression evaluation for C++ in LLDB
  - <https://werat.dev/blog/blazing-fast-expression-evaluation-for-c-in-lldb/>
- R. Isemann “Better C++ debugging using Clang Modules in LLDB”
  - <https://www.youtube.com/watch?v=vuNZLLHhy0k>