THE VERY SOUL OF C++

Language standard, generic programming and program behavior

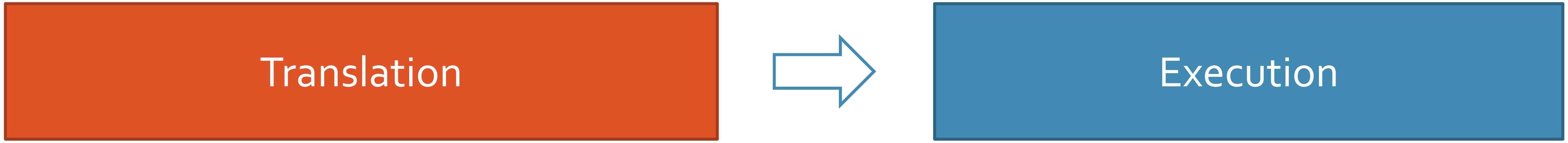
K. Vladimirov, Syntacore, 2025
mail-to: konstantin.vladimirov@gmail.com

About this course

- The Master's course is taught over two semesters. There is an exam at the end of each semester.
- During the course, assignments can be completed. Based on their results, many exam grades can be assigned automatically.
- Slides will appear here: https://sourceforge.net/projects/cpp-lects-rus/files/cpp-postgraduate-2025-26
- Code examples on github: https://github.com/tilir/cpp-masters

The C++ programming language

- General purpose, statically typed, **statically compiled**.
- There are two main phases.
  - The translation phase.
  - The execution phase of the translated program.



- The language standard determines if the program can be translated.

The language standard

- A programming language is an agreement between the programmer and the compiler developer.
- As such, it is documented in the language standard.
- It is the standard, not a specific implementation, that is the final and decisive argument on how a program is translated.
- The current C++ standard is ISO/IEC 14882:2023, adopted in 2023 by the International Organization for Standardization (ISO).
- Detailed information is available at `https://isocpp.org/std/the-standard`

Syntax and semantics

- The language standard is a list of diagnosable rules.
- Rules can be syntactic (which can be checked by grammar).

```
template<int i = 3 < 4> struct S; // syntax violation
```

- And semantic, which are sometimes difficult to check.

```
int foo(int); // foo shall be defined somewhere
int bar(int x) { return foo(x); }
```

- An implementation that satisfies the standard should ideally either translate the program or issue a diagnostic.

Normative references

- From edition to edition, absolute section numbers can change.
- C++17 overloading is Chapter 16.
- C++20 overloading is Chapter 12.
- Therefore, the standard uses symbolic references. In both these documents, the section is labeled as [over], and subsections like C++17, 16.1 or C++20, 12.2 are labeled as [over.load].
- A typical reference looks like [C++17, over] or simply [over] if the standard is not important, if the latest one is meant, or if it is clear from the context.

https://en.cppreference.com/w/cpp/links

Let's start with a bit of C code

- How would you write a program that calculates the position of the most significant set bit in a number?
- In the C language, you would probably do it something like this:

```c
int find_msb(MSB_TYPE value) {
  int maxbit = sizeof(MSB_TYPE) * CHAR_BIT;
  for(int i = 0; i < maxbit; ++i){
    int bitnum = maxbit - i - 1;
    MSB_TYPE mask = (MSB_TYPE) 1 << bitnum;
    if ((value & mask) == mask) return bitnum;
  }
  return -1; // nothing found
}
```


https://godbolt.org/z/b87916rWf

We can do better

- For those types that support `__builtin_clz`

```
int find_msb(unsigned value) {
  int maxbit = sizeof(unsigned) * CHAR_BIT;
  int leading_zeros = __builtin_clz(value);
  int msb_position = maxbit - 1 - leading_zeros;
  return msb_position; // nothing found is -1
}
```

- This is much faster but much less portable between compilers.
- And the worst part — in the C language, it is very difficult to achieve both.

https://godbolt.org/z/q1aTs6f88

Normal vector of development

- From common idioms to special assembly instructions.

```
find_msb: li      a5,31
          clzw    a0,a0    # -march=rv64gc_zbb
          subw    a0,a5,a0
          ret
```

- From macros to generic programming.
- «Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer» © Bjarne Stroustrup
- From compiler intrinsics to standard library functions.

```
__builtin_clz → template<class T> int countl_zero(T x);
```


https://en.cppreference.com/w/cpp/numeric/countl_zero.html

Let's try C++

- We will be learning C++23, occasionally touching on C++26. First attempt is occasionally **ill-formed**.

```
template<typename T> int find_msb(T value) {
  int total_bits = sizeof(T) * CHAR_BIT;
  int leading_zeros = std::countl_zero(value);
  int msb_position = total_bits - 1 - leading_zeros;
  return msb_position;
}

assert(find_msb(1) == 0);
```

```
error: no matching function for call to 'countl_zero'
  int leading_zeros = std::countl_zero(value);
```


https://godbolt.org/z/3dn8nqGoh

Well-formed and ill-formed

- A program that cannot be translated is called ill-formed.

```
int main() {
  72057594037927936; // well-formed
}
```

```
int main() {
  4611686018427387904OO; // ill-formed [lex.icon]
}
```

- Sometimes a program is ill-formed, but the compiler cannot diagnose it. In this case, we assign it the status IFNDR, relying on the assembler or the linker.

https://godbolt.org/z/4E1znP9hY

Make type unsigned

- When we generalize something with respect to type, we need operations on types, not just on values.

```
template<typename T> int find_msb(T value) {
  using U = std::make_unsigned_t<T>;
  int digits = std::numeric_limits<U>::digits;
  int leading_zeros = std::countl_zero(static_cast<U>(value));
  int msb_position = digits - 1 - leading_zeros;
  return msb_position;
}

assert(find_msb(1u << 31) == find_msb(-1) == 31);
```


https://godbolt.org/z/z3EPWP99x

Make auto local variables

- But better do not touch non-template parts of the signature.

```
template<typename T> int find_msb(T value) {
  using U = std::make_unsigned_t<T>;
  auto digits = std::numeric_limits<U>::digits;
  auto leading_zeros = std::countl_zero(static_cast<U>(value));
  int msb_position = digits - 1 - leading_zeros;
  return msb_position;
}
```

- This way you can be sure, that zero still results in -1.

https://godbolt.org/z/MMMdovj9M

What about user-defined types?

- We would like to make overloads for non-integral types.

```
struct ModularInt { /* something weird */ };
```

- To do this, we must behave politely in the main version.

```
int find_msb(std::integral auto value) {
  using U = std::make_unsigned_t<decltype(value)>;
  auto digits = std::numeric_limits<U>::digits;
  auto leading_zeros = std::countl_zero(static_cast<U>(value));
  int msb_position = digits - 1 - leading_zeros;
  return msb_position;
}
```


https://godbolt.org/z/45988E55c

Problem with unsigned char

```
int find_msb(std::integral auto value) {
  using U = std::make_unsigned_t<decltype(value)>;
  auto digits = std::numeric_limits<U>::digits;
  auto leading_zeros = std::countl_zero(static_cast<U>(value));
  auto msb_position = digits - 1 - leading_zeros;
  return msb_position;
}

// ok
template int find_msb<unsigned>(unsigned);

// slightly worse (see asm)
template int find_msb<unsigned char>(unsigned char);
```


https://godbolt.org/z/Y6MaKK5rr

Problem with unsigned char

```
int find_msb<unsigned char>:
        movzx   edi, dil
        mov     eax, 31
        lzcnt   edx, edi
        sub     eax, edx
        test    edi, edi
        mov     edx, -1
        cmove   eax, edx
        ret
```

```
int find_msb<unsigned int>:
        mov     eax, 31
        lzcnt   edi, edi
        sub     eax, edi
        ret
```

- The compiler, even for 8-bit integers, uses 32-bit registers and therefore builds additional code to handle zero at the input.

The grandmaster sacrifices the exchange

- We can say that the function's behavior for input zero is undefined.

```
// find_msb is undefined for input zero
int find_msb(std::integral auto value) {
  if (value == 0) std::unreachable();
  using U = std::make_unsigned_t<decltype(value)>;
  auto digits = std::numeric_limits<U>::digits;
  auto leading_zeros = std::countl_zero(static_cast<U>(value));
  int msb_position = digits - 1 - leading_zeros;
  return msb_position;
}
```

- This will allow the compiler to optimize it much better.

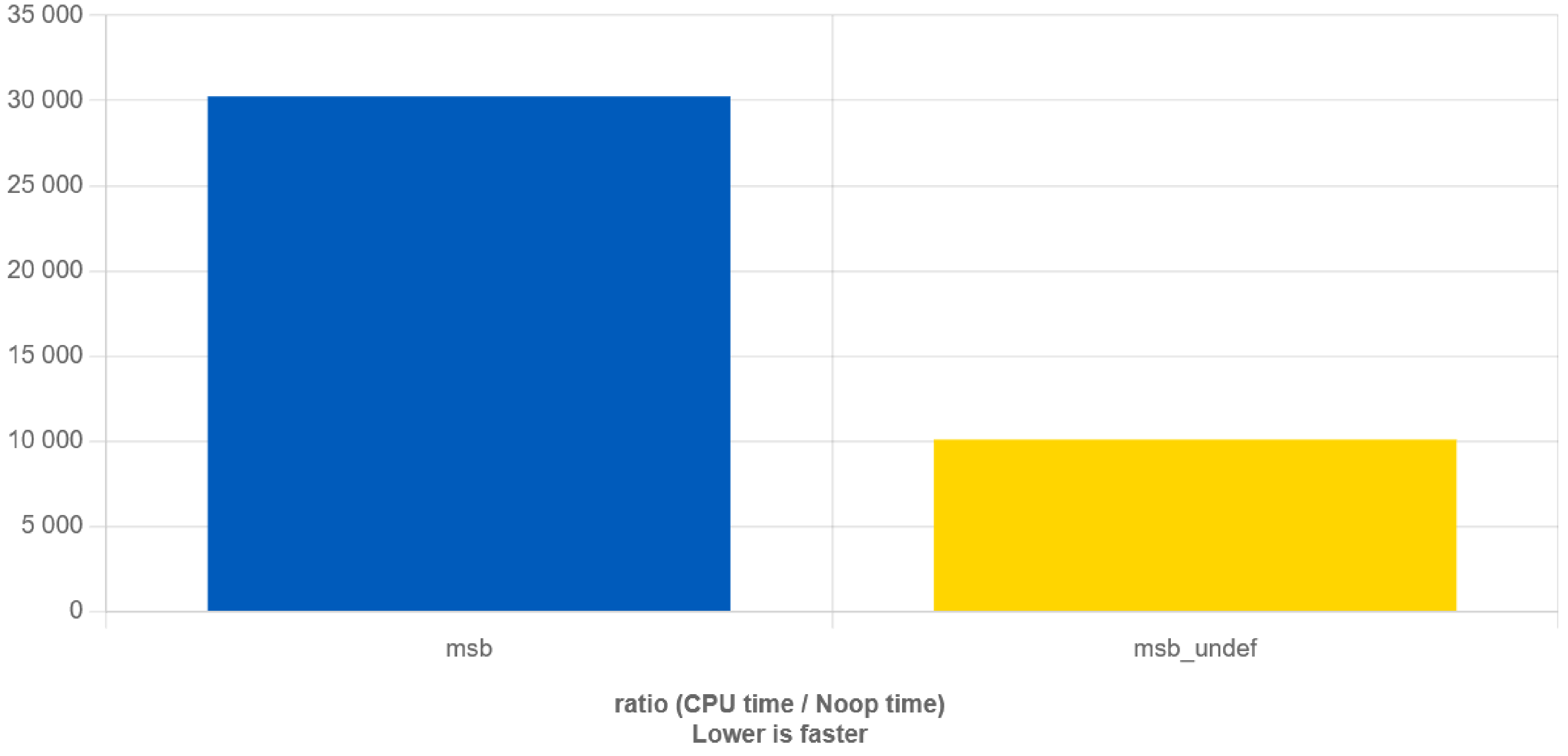
https://godbolt.org/z/ssPWezTW9

Assembler obviously improved

```
int find_msb<unsigned char>:
        movzx   edi, dil
        mov     eax, 31
        lzcnt   edx, edi
        sub     eax, edx
        test    edi, edi
        mov     edx, -1
        cmove   eax, edx
        ret
```

```
int find_msb<unsigned char>:
        movzx   edi, dil
        mov     eax, 31
        lzcnt   edi, edi
        sub     eax, edi
        ret
```

Btw, always benchmark things



https://quick-bench.com/q/u-3Oyqpbi2-4FPf7j-kuBpRPDHE

Kittens are in danger

- We can say that the function's behavior for input zero is undefined.

```
// find_msb is undefined for input zero
int find_msb(std::integral auto value) {
  if (value == 0) std::unreachable();
  using U = std::make_unsigned_t<decltype(value)>;
  auto digits = std::numeric_limits<U>::digits;
  auto leading_zeros = std::countl_zero(static_cast<U>(value));
  int msb_position = digits - 1 - leading_zeros;
  return msb_position;
}
```

- This will allow the compiler to optimize it much better. But it can cause the death of your kitten if the condition is violated.

https://godbolt.org/z/1eEbq6Pxr

Document intent with contract

- We can state, that the function's behavior for input zero is undefined.

```
int find_msb(std::integral auto value) pre(value != 0) {
  if (value == 0) std::unreachable();
  using U = std::make_unsigned_t<decltype(value)>;
  auto digits = std::numeric_limits<U>::digits;
  auto leading_zeros = std::countl_zero(static_cast<U>(value));
  int msb_position = digits - 1 - leading_zeros;
  return msb_position;
}
```

- We won't go into detail here, as we're in a hurry to get back to the dead kittens.

Abstract machine

- The semantic descriptions in this document define a parameterized nondeterministic abstract machine [C++, intro.abstract].

```
int zero() { // external linkage
  int i = 0;
  int j = i * i + i * 3 + 42; // no side effects
  return i;
}
```

- The entire language standard document defines the abstract machine.
- The compiler does not generate code for it, but it considers its behavior during optimizations.

https://godbolt.org/z/acKec5Y63

Volatile qualifier

- A special qualifier is needed to describe unpredictable effects in the abstract machine.

```
int zero() { // external linkage
  int i = 0;
  volatile int j = i * i + i * 3 + 42; // unknown side effects
  return i;
}
```

- volatile can be read as "may change unpredictably".

```
const volatile int i = 0;
```

- An initializer is required here (which is generally ignored) because const requires it.

https://godbolt.org/z/KxG6PjTj6

What the translator can do: the as-if rule

[intro.abstract] […] conforming implementations are required to emulate (only) the observable behavior of the abstract machine.

- What constitutes observable behavior?
  - **Accesses through volatile glvalues**
  - Data written into files
  - The input and output dynamics of interactive devices
- The compiler is allowed to do anything with the program as long as the observable behavior remains the same.

We must be conservative

```
int nonzero(int i) // external linkage
{
  return i * i + i * 3 + 42; // no side effects, but...
}
```

- The compiler does not know whether this expression will be used later to produce a side effect.
- The scope within which the compiler can track all connections is called a translation unit.
  - For the C language, it is a file.
  - For the C++ language, we will have a lecture about that.

Discussion

- Will there be a side effect here?

```
volatile std::nullptr_t a = nullptr; int *b; b = a;
```

```
// clang
mov     qword ptr [rsp - 8], 0
xor     eax, eax
```

```
// gcc
mov     QWORD PTR [rsp-8], 0
mov     rax, QWORD PTR [rsp-8]
xor     eax, eax
ret
```


https://godbolt.org/z/rP7eeP4EK

How conservative?

```
int foo(int *a, double *d, int n) {

  for (int i = 1; i < n; ++i) {
    d[2] = 1.0;
    a[i] += i * i;
  }

  return d[a[0]];
}
```


https://godbolt.org/z/W5b1vMboz

Strict aliasing

```
int foo(int *a, double *d, int n) {
```

- [basic.lval] If a program attempts to access the stored value of an object through a glvalue whose type is not similar to one of the following types the behavior is undefined:

A. the dynamic type of the object

B. a type that is the signed or unsigned type corresponding to the dynamic type of the object

C. a char, unsigned char, or std::byte type

- There is a very dubious option -fno-strict-aliasing that blocks the compiler's strict aliasing-based optimizations.

Details of the abstract machine

- The abstract machine is:
- Parameterized.

```
int nbits = std::numeric_limits<unsigned int>::digits;
```

- Non-deterministic.

```
int is_equal = (+"abc" == +"abc");
```

- Not defined everywhere (and this is very interesting).

Undefined behavior is important

- Undefined behavior gives the optimizer free rein.

```
int foo(int *a, double *d, int n);
int arr[10] = { /* some initializer */ };
double *d = (double *)(&arr[0]);
foo(arr, d, 10); // this function will be optimized
                 // as if arr and d do not overlap!
```

- No one will warn you.
- For the optimizer, undefined behavior **does not exist**.

Undefined behavior is dangerous

- [intro.abstract] A conforming implementation executing a well-formed program shall produce the same observable behavior [...].
  However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (**not even with regard to operations preceding the first undefined operation**).

```
int k, satd = 0, dd, d[16];

/* .... more code here .... */

for (dd = d[k = 0]; k < 16; dd = d[++k]) // how do you think?
  satd += (dd < 0 ? -dd : dd);
```

The compiler is blind to UB

- The compiler always behaves as if UB does not exist.

```
int f() {
  int i; int j = 0;
  for (i = 1; i > 0; i += i)
    ++j;
  return j;
}
```

- A legitimate optimization of this code is again an infinite loop.

```
f():
.L2:  jmp     .L2
```


https://godbolt.org/z/qWYfaaozE

Freedom of optimization around UB

- The whole concept of "use undefined C behavior to change code generation" is complete and utter BS. It's wrong. It's stupid. And a compiler shouldn't do it. (c) Linus Torvalds

```c
int ubranch(int n) {
    int k = 1;
    switch(n) {
      case 0: k = 7; break;
      case 2: k = 0; break;
      case 9: k = 4; break;
    }
    return n / k;
}
```

```
ubranch(int):
         li      a1, 9
         beq     a0, a1, .LBB0_5
         li      a1, 2
         beq     a0, a1, .LBB0_4
         li      a1, 1
         bnez    a0, .LBB0_6
         li      a1, 7
         divw    a0, a0, a1
         ret
.LBB0_4: divw    a0, a0, zero
         ret
.LBB0_5: li      a1, 4
.LBB0_6: divw    a0, a0, a1
         ret
```

https://godbolt.org/z/Y9ohG75cc

https://github.com/llvm/llvm-project/pull/137741

Freedom of optimization around UB

- The whole concept of "use undefined C behavior to change code generation" is complete and utter BS. It's wrong. It's stupid. And a compiler shouldn't do it. (c) [Linus Torvalds]()

```c
int ubranch(int n) {
    int k = 1;
    switch(n) {
      case 0: k = 7; break;
      case 2: k = 0; break;
      case 9: k = 4; break;
    }
    return n / k;
}
```

```asm
ubranch(int):
         li      a1, 9
         beq     a0, a1, .LBB0_2
         li      a1, 1
         beqz    a0, .LBB0_3
         j       .LBB0_4
.LBB0_2: li      a1, 4
         bnez    a0, .LBB0_4
.LBB0_3: li      a1, 7
.LBB0_4: divw    a0, a0, a1
         ret
```

https://godbolt.org/z/bjrd5dWfT

https://github.com/llvm/llvm-project/pull/137741

A golden opportunity to score a 10

- We will study C++. Fortunately, most industrial compilers are written in C++.
- Integrate into any industrial compiler a patch that removes code along paths leading to UB, which was not previously removed.
- For example, be the first to exploit the following UB (via Vladislav Belov).

```
void foo(short const * const p);

int test2(short p1) {
  const short p2 = p1; foo(&p2); // UB if foo changes p2

  if (p1 == p2) return 14;
  return 42; // This branch to be disregarded
}
```


https://godbolt.org/z/bKKe437ch

Let me scare you now

- Suppose you wrote code that "protects you from UB".

```
int foo(int *a, int base, int off) {
  if (off > 0 && base > base + off) return 42;
  return a[base + off];
}
```

- In the assembly, you suddenly see a strange thing: the compiler has erased all the checks.

```
foo(int*, int, int):    add     esi, edx
                        movsxd  rax, esi
                        mov     eax, dword ptr [rdi + 4*rax]
                        ret
```


https://godbolt.org/z/scdWKT9T3

The compiler removed my code…

- It can do this for two reasons.
- Due to the as-if rule, if your code did not affect side effects.
- If your code was on a path that also contains undefined behavior.

```
int foo(bool c) { // foo(true) == 42
  int x, y;
  y = c ? x : 42;
  return y;
}
```

- This irritates many people, and in C++26 a new type of behavior was defined.

https://godbolt.org/z/5xzhbGGWz

Erroneous behavior (C++26)

```
char foo() {
  char a;
  char b [[indeterminate]];

  char c = a + 1; // erroneous, not undefined from C++26
  char d = b + 1; // still UB

  return c + d;
}
```

- well-defined behavior that the implementation is recommended to diagnose [defns.erroneous]

Erroneous values (C++26)

```
unsigned char c;
unsigned char d = c;  // no erroneous behavior,
                      // but d has an erroneous value
assert(c == d);       // holds, both integral promotions
                      // have erroneous behavior
```

- When storage for an object with automatic or dynamic storage duration is obtained, the bytes comprising the storage for the object have the following initial value:
  - If the object has dynamic storage duration, or [...] marked with the [[indeterminate]] attribute, the bytes have **indeterminate values**;
  - otherwise, the bytes have **erroneous values**, where each value is determined by the implementation independently of the state of the program.

Homework assignment

- $[HW1.1][1]$ Justify the situation with volatile nullptr_t using the standard.

```
volatile std::nullptr_t a = nullptr; int *b; b = a;
```

- $[HW1.2][1]$ Find another elegant example where dereferencing a null pointer is valid, before your classmates do.

```
int *a = nullptr;
std::println("{}", typeid(*p).name()); // not even exception
```

- $[HW1.3][1]$ Find how to write proper protection code for

```
int foo(int *a, int base, int off) { return a[base + off]; }
```

Bibliography

- ISO/IEC, "Information technology – Programming languages – C++", ISO/IEC 14882: 2023
- Bjarne Stroustrup, The C++ Programming Language (4th Edition), 2013
- Richard Powell, "Intro to the C++ Object Model", CppCon, 2015
- Chandler Carruth, "Garbage In, Garbage Out: Arguing about Undefined Behavior...", CppCon, 2016
- Piotr Padlewski, "Undefined Behaviour is awesome!", CppCon, 2017
- Bob Steagall, "Back to Basics: The Abstract Machine", CppCon, 2020
- Herb Sutter, "Three Cool Things in C++26: Safety, Reflection & std::execution", C++ on Sea, 2025

Linus quotes some article

- Popescu, Lopes, "Exploiting Undefined Behavior in C/C++ Programs for Optimization: A Study on the Performance Impact"
- The authors of the article examine **24 benchmarks** using LLVM 16 on **x86 and ARM** architectures.
- Then, using options, they enable and disable 18 different UB-based optimizations.
- Their (perfectly normal in itself) work cannot serve as an argument in this kind of debate. Compilers optimize **millions** of applications, on **tens** of architectures.
- For each UB-based optimization, we can fairly easily create a winning benchmark.
- A good argument would be a benchmark that shows code degradation due to a particular UB-based optimization.

https://web.ist.utl.pt/nuno.lopes/pubs/ub-pldi25.pdf