A Low Level Language with Precise Integer Types

Ian Shehadeh IRHSHEHADEH@SMCM.EDU

Department of Math and Computer Science
St. Mary's College of Maryland
St. Mary's City, Maryland, USA

Abstract

We present *Howlite* a language targeting RISC-V, with a similar level of abstraction to C. Howlite uses a single scalar type, *integer*, which allows users to specify exactly the set of values allowed. Collection types are checked with a simple, structural bi-directional type checker.

Keywords: programming language

1 Programming Language Concepts and Motivation

Programming Language is a broad term, generally, it's a text-based format for expression computation. Although most programs are written by software engineers, and most programming languages are designed with software engineers in mind, this by no means makes them a homogenous group. After a program is written it's read by a machine (the compiler or interpret) and people - possibly just the original author, or others they collaborate with. To satisfy both audiences it must be clear two ways: formally, so the machine can produce consistent and accurate results, and legible, the original author's intent should be clear.

How we make a programming language clear depends entirely on what it's meant to accomplish, to demonstrate, consider *Hedy*, Hermans (2024), and *Go*, (2024).

```
print Hello!
ask What is your name?
```

Example 1: Hedy (English)

```
                !Hello قول
?What is your name اسأل
```

Example 2: Hedy (Arabic)

```
func indexOf(str string, c rune) int {
  codepoints := []rune(str);

  for i := 0; i < len(codepoints); i++ {
    if codepoints[i] == c { return i; }
  }

  return -1;
}
```

Example 3: Go

1.1 Hedy

Hedy Hermans (2024) is a programming language for teaching programming.

The language avoids symbols, instead using keywords, which are generally easier for students to remember. With so much of the language being textual, Hedy is fully translated to a large set of languages - 47 at the time of writing. Both Hedy programs above do the same, but they're keywords are in different languages. Hedy also allows programmers

to see and hear the results of their work: it has easily accessible functionality for playing music and drawing graphics. Those features are typically implemented as libraries, for most programming languages, since the have a relatively narrow application. But, it takes a lot of general knowledge to have the computer *do* anything interesting; when first learning, it's helpful to be able to make the computer **do** something, not just report results in the terminal.

1.2 Go

Go was an answer to problems with the software infrastructure at Google (Pike (2012)). It's designed to be used in large, long-lived software projects. There's a focus on clear syntax and semantics: no matter who wrote the code, what it does, and how it does it should be clear to any programmer. It also comes bundled with tools to help keep programs up to date and consistent. Unused variables and imports are disallowed. Although it supports first class functions, it's largely imperative. Programmers are forced to deal with the inherent complexity of things like string encoding up front, as seen in Example 3.

1.3 Howlite's Purpose

```
func to_num(c: char): -1 | 0..9 {
  if ('0' <= c && c <= '9') {
    c - '0'
  } else { -1 }
}
```

Example 4: Howlite

Howlite is an expiremental language for writing programs that necessitate little abstraction over they machines they control. The project's goal is to answer the question, *How can we create an expressive type system without limiting a programmer's control of the hardware?*

2 The Programming Language

Memory safety in systems programming languages has garnered a lot of attention in the last several years. A compiler that enforces strict rules on an object's lifetime and mutability is helpful in large projects, especially when security is a top concern. Checking these properties at compile time allows the compiler to omit parts of its runtime, like a garbage collector, while providing similar gaurentees.

These innovations in language design fail to directly address a class of problems where direct memory manipulation is essential. These problems force the programmer to fully disable the compiler's checks, or encourage awkward solutions which trade clarity for small gaurentees.

Howlite aims to address these problems. Howlite is not a language to write a web server, it is not for writing applications, it isn't even a language for writing programming languages. It is a language for writing a single module for a very specific data structure, wrapped in a python library. It is a language for writing a boot loaded, or the entrypoint to a kernel. The compiler does not impose strict requirements on how the programmer manages memory, or accesses data. Instead, the type systems gives a rich set of tools, allowing one to set their own constraints.

2.1 Overview

The most notable feature of Howlite is the type system. The type system is structural, and closely tracks the value of integers. You can declare types which only accept specific integers, or ranges. Types are compared based on their compatability, not by name.

```
type Boolean = 0 | 1;

let true: Boolean = 0;
let false: Boolean = 1;
```

Example 5: Boolean Type

To better understand the language, this section will walk through the process of definining a function to get the index of a character from an ASCII string.

```
type char = 0..127;
type u32 = 0..0xffffffff;
type NatI32 = 0..0x7fffffff;
```

Example 6: Range Types

First we define a character as any number between `0` and `127`, or the base set of ASCII characters. Next is the definition of standard 32-bit integer, this will be used to index the array. Finally, we define the a variant of `i32` that is only positive. We'll use this type to represent the index, which can't be negative.

```
func index_of[LenT: NatI32](str: &[char; LenT], c: char): 0..Max[LenT] | -1
```

Example 7: Function Signature

This function is generic, the `[LenT: NatI32]` section indicates, that for any subset of the positive, signed, 32-bit integers, there is an instance of `index_of`. Whatever that type is, it's refered to as call it `LenT`, within the context of this function.

Moving on to the parameter list, notice the type of `str` is `&[char; LenT]`. This `&[...]` is a special type called a *slice* (also know as a fat pointer). It is implemented as a pointer and length pair, and practically it functions like an array. These types are common, they're primitives in Rust, Go, and Zig. Although it's not a primitive type, the C++ STL's `std::span` is a similar data structure. What sets our slice type apart is that the type of the length can be set. So, for example say we take a slice of some ASCII string, from index 3 to 10, the result would have the type `&[char; 7]`.

By using a generic parameter, `LenT`, then giving `str` the type `&[char; LenT]`. We can be certain this function only works on a string of length less than or equal to 0x7fffffff. Since it's impossible to find a character outside of those bounds, we know the return type can't exceed the maximum value of `LenT`, if no character is found then we return `-1`.

Finally, the body of this function likely looks familiar to C programmers, with some minor syntactic changes. Variables are declared with `let`, `mut` indicates that we can change the value after initialization. All expressions (including `if` statements and blocks) have values. The value is equal to the value of the last line in the block, if it omits a semicolon (`;`), or `unit` otherwise.

```
{
  let mut i: u32 = 0;
  while i < str.len {
    if str[i] == chr {
      return i;
    }

    i = i + 1;
  }

  -1
}
```

Example 8: Expression

Some care must be taken to make sure we satisfy the return type. How can the compiler be certain `i` is always a subset of `0..Max[LenT]`, since `u32` certainly exceeds `LenT`. The condition `while i < str.len`, narrows `i`'s type from `u32` to `0..LenT-1`. This means within the body of that loop, `i` can be used as if it had the type `0..Max[LenT]-1`. Arithmetic will modify this type: after running `i = i + 1`, `i`'s narrow type has change to `1..Max[LenT]`. If we changed the code to check some other condition, for example `chr < str.len`, this wouldn't compile.

3 Syntax Design

Howlite's syntax prioritizes familiarity, ease of parsing, and clarity. The syntax should be immediately familiar to anyone who knows another C-like language. The grammar is context free, so it can easily be expressed using a parser generator. The language's syntax should give programmer's the tools to express their intent.

```
func fib(n: u32): u32 {
    if n == 0 { 0 }
    else if n == 1 { 1 }
    else {
      fib(n - 1) + fib(n - 2)
    }
}
```

Example 9: Recursive Fibonacci

For example, flow control constructs, like if statements may have a value. This allows the programmer to clearly show a variable's value is the result of some condition. In order to make tooling easier to write, we prioritize createing an unambigous grammar, with no constructs that require unbounded look-ahead.

3.1 Familiarity

Howlite code should be recognizable to C programmers. For this reason, we use curly braces ("{" and "}") to denote blocks of code. We use familiar imperative keywords: "if", "else", and "while", and mathmatical expressions follow typical infix notation. Howlite differs from C in that it requires a sigil character or keyword before beginning a new construct. Types do not lead in variable assignments or functions. Instead we use the "let" or "func" keywords, respectively. This simplifies parsing, since we know what type of statement or expression will follow, similarly, type ascriptions are always prefixed with `:`. These keywords and symbols were decided by surveying popular languages during design. For example, "let", and `:` come from TypeScript, while "func" is a keyword in Go.

3.2 Ease of Parsing

A small, easily parsed grammar is valuable because it makes implementing tooling easier. Anything from simple syntax highlighting in *Emacs* to an auto-formatter or linter dramatically easier to implement when parsing the language isn't a significant hurdle.

Howlite's syntax is expressable in an LR grammar. Consequently, the grammar is unambiguous. While writing the grammar, we aimed to reduce look ahead as much as possible. For example, function's type parameters are written `index_of[:u32](...)`, which disambiguates the use of `[...]` from array access.

3.3 Clarity

Here, we use clarity to mean the ease of understanding a program's behavior. If a program is clear, then the author's original intent should be easily understood by someone familiar with the language. The author of a program is responsible for making their intent clear; the syntax should guide their choices, and give them the tools to express their intent.

We optimize clarity by keeping tokens consistent, for example colon (`:`) is almost always a way to give *something* a type, whether that thing is an expression, variable, or a field of a data structure. However, we don't sacrifice familiarity for consistency. Languages like C, C++, Java, Go, and more use curly braces for both structure declarations and statement blocks, so we follow suit.

Giving programmers tools to express their intent extends beyond syntax, but the features a language's syntax emphasizes plays an important role in guiding the programmer. Being a low-level language, we want to emphasize precisely what the machine is doing. Howlite programs are written in an imperative style, we expect the programmer to use mutable state, but discourage it when unessary by making it opt-in, via the `mut` keyword. We also omit short hand syntax or functions for functional operations, like transforming the content of an array. While these operations are convient, they paper over important, they can paper over important details like memory allocation.

4 Type Checking

We optimize clarity by keeping tokens consistent, for example colon (`:`) is almost always a way to give *something* a type, whether that thing is an expression, variable, or a field of a data structure. However, we don't sacrifice familiarity for consistency. Languages like C, C++, Java, Go, and more use curly braces for both structure declarations and statement blocks, so we follow suite.

Howlite's implements a simple bi-directional type checker [Dunfield and Krishnaswami (2020)]. Every node in the AST is given a type. An AST node's type is typically derived from it's children's types, through a process called *synthesis*, we call these types *synthesized types*. Many constructs in the language must be ascribed types by the programmer: variables declared with "`let`", function parameters, and return values. Types which are declared explicitly are called *assumed* types.

```
let a: Uint32 = 1;
```

Listing 1: Simple Let statement

Here, `Uint32` is the assumed type of `x`. Where ever `x` is referenced, we can consider it of type `Uint32`. The literal `1` has no assumed type. Instead, we synthesize a type for `1` by following a set of rules. For literals, this rule is simple: *for a literal scalar N the synthesized type is* $\{N\}$. As expressions grow, synthesizing types becomes more complicated.

4.0.1 Type Checking an AST

To better illustrate this process, we'll walk through synthesizing a tree.

```
func average(x : 0..10, y : 0..10, z : 0..10) : 0..10 {
  (x + y + z) / 3
}
```

The function parameters: `x`, `y`, and `z` have each been given the assumed types `UInt32`. An assumed type is analogous to the the statement "no matter the value of `x`, we can always assume it is a `UInt32`". The function's assumed return type is `UInt32`. This allows any caller to treat the expression `average(a, b, c)` as a `UInt32`, even if the operations performed by the function are unknown. An assumed type is a promise; it allows the references to entity to *assume* the type of that entity, without knowing anything else about it.

To illustrate how these assumed types interact with synthesized types, we'll manually type check the function.

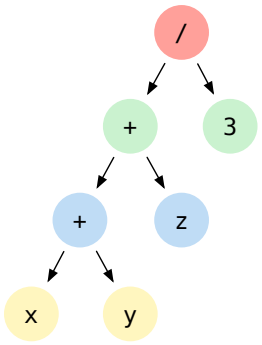
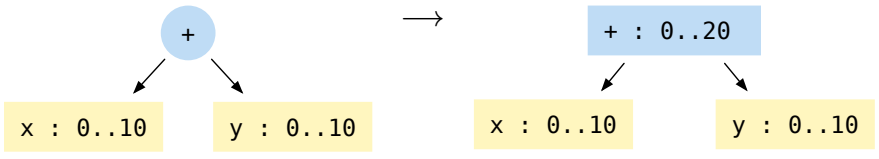


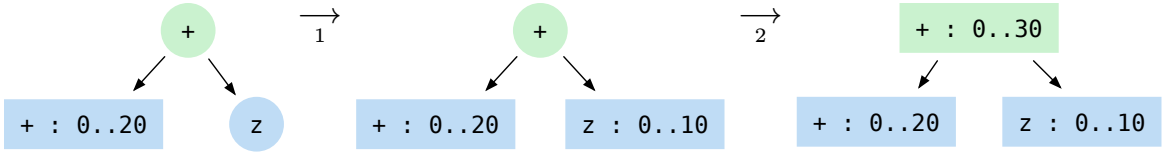
Figure 1: AST

The funtion body, `(x + y + z) / 3`, has the syntax tree seen in Figure 1. The type checker works bottom-up, left-to-right. So, we begin with the leaves of the tree: `x`, and `y`. Identifier AST node's synthesized type is the assumed type of the symbol they include. So `x` is synthesized to type `0..10` (the assumed type of `x`), and `y` is synthesized to type `0..10` (the assumed type of `y`).

This information is added to the tree, and we reference it synthesize `+`. An operator node's synthesized type is constructed by applying the given operation to the synthesized types of each operand. Types may be constructed using arithmetic operations, this process will be defined more formally in Section 4.1. For now, take for granted that `0..10 + 0..10 : 0..20`.

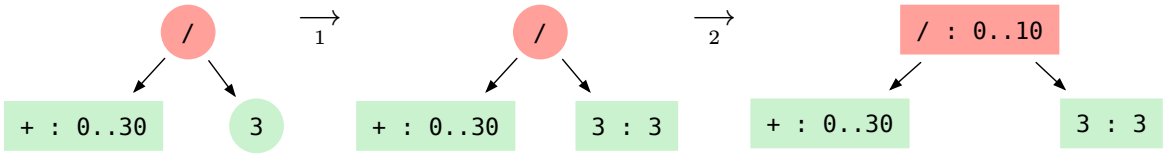


Now, we move up the tree, to synthesize the right hand side of `+`, then finally `+` itself.



In (1) we synthesize the node's type from the assumed type of `z`. In (2) we used this information, and the type of `+` to synthesize a type for `+`.

Finally, we again move up the tree, now to `/`.



Due to the the functions return value, the assumed type of the body is `0..10`. Function body's type is synthesized based on the possible return values. So the synthesized type of this function's body is the the type of `/`.

Type checking is the process of comparing assumed and synthesized types. If a synthesized type is not a subset of the assumed type, then a type error is attached to that node.

4.1 Scalars

There is a single scalar type in Howlite, this simplifies the type checking by condensing many cases into a single, generic case. There are no distinct enumerable types, true boolean types, or even a unit type in the language. Instead of distinct types, we have the scalar type "Integer" (floating point numbers are out of scope). A scalar may be any set of Integers.

4.1.1 Synthesis of Scalars

As seen above, a scalar may be systhensized from a single value, for example the type of -5 is $\{-5\}$. We can also construct new scalars using arithmetic operations:

Given a scalar type $T = \{t_1, t_2, t_3 ... t_n\}$, where $\forall i : t_i \in \mathbb{Z}$, and a scalar type $U = \{u_1, u_2, u_3 ... u_n\}$ where $\forall j : u_j \in \mathbb{Z}$. (i.e T, U are subsets of the integers). We can construct the following types:

- $T \times U = \{tu : \forall t \in T, \forall u \in U\}$
- $T + U = \{t + u : \forall t \in T, \forall u \in U\}$
- $T - U = \{t - u : \forall t \in T, \forall u \in U\}$
- $T \div U = \{t \div u : \forall t \in T, \forall u \in U\}$

For example, given $T = \{1, 2, 3\}$ and $U = \{-5, -7\}$, we'd compute the following:

- $T \times U =$
  $\{1(-5), 2(-5), 3(-5), 1(-7), 2(-7), 3(-7))\} = \{-5, -10, -15, -7, -14, -21\}$

- $T + U = \{1 + -5, 2 + -5, 3 + -5, 1 + -7, 2 + -7, 3 + -7) = \{-4, -3, -2, -6, -5, -4\}$
- $T - U = \{1 - (-5), 2 - (-5), 3 - (-5), 1 - (-7), 2 - (-7), 3 - (-7)) = \{6, 7, 8, 9, 10\}$
- $T \div U = \{1 \div (-5), 2 \div (-5), 3 \div (-5), 1 \div (-7), 2 \div (-7), 3 \div (-7)) = \{0\}$

4.1.2 Storage Classes

Scalar types belong to a *storage class* that identifies how they are encoded in memory. Storage classes are organized by size, whether or not they include a sign bit. The signed storage classes are `s8`, `s16`, `s32`, `s64`, and the unsigned are `u8`,`u16`,`u32`,`u64`. Going forward, we will identify the storage class of a scalar `T` using the notation `u32[T]`.

The storage class of a number influences how arithmetic and bitwise operations behave on the inner type.

4.1.2.1 Unsigned Storage Classes

given a storage class `uN`, where N is the width in bits, and variables `a : uN[T]`, and `b : uN[T]`

- $a + b = (a + b) \bmod 2^N$
- $a - b = 2^N - |a - b| \bmod 2^N$
- $a * b = (a * b) \bmod 2^N$
- $a \div b = (a - (a \bmod b)) \div b$ (i.e. division is always rounded down)
- $\sim a = (2^N - 1) - a$
- TODO other bitwise ops defined in terms of the above operations
- TODO except xor, maybe?

4.1.2.2 Signed Storage Classes

given a storage class `uN`, where N is the width in bits, and variables `a : sN[T]`, and `b : sN[T]`

- $a + b = (a + b) \bmod 2^N$
- $a - b = 2^N - |a - b| \bmod 2^N$
- $a * b = (a * b) \bmod 2^N$
- $a \div b = (a - (a \bmod b)) \div b$
- $\sim a = (2^N - 1) - a$
- TODO other bitwise ops defined in terms of the above operations
- TODO except xor, maybe?

4.2 Narrowing

A variable's type may be narrowed based on the result of a boolean expression.

```
let x: UInt32 = /* ... */;
let y: &[Char; 0..100] = /* ... */;

if x <= 100 {
  print(y[x]);
}
```

Within this if-statements body, the synthesized type of `x` has been narrowed to `0..100`.

This is achieved by assigning *implications* to values. Here, we have `(x <= 100) : 0 | 1`, the value `1` is assinged the implication `x : 0..100`, and the value `0` is assigned `x : 101..0xffffffff`.

A type carrying implications appears in a conditional (at the time of writing, just `if` statements) then the implications of a value, a, are applied within a block if the conditional gaurentees the expression had the value a before entering the block.

5 Disjoint Integer Sets

Integer sets are used throughout the type checker. The semantics of our type system (see Section 4.1) require these sets implement arithmetic operations in addition to usual set operations like union, intersect, etc.

Representations of sparse sets in memory is a well studied topic, with efficient solutions for many use cases. Most of the work we found focuses on storing collections of integers, but performing operations on them isn't well optimized.

To date, we have not found an efficient method of computing the operations laid out in Section 4.1 in the general case. Instead, we've focused on optimizing operations often performed by the programmer, while offering them ways to bypass strict integer checks when required.

Internally, we use 3 set representations, *Stripe Sets*, *Small Sets* and *Continguous Sets*

5.1 Stripe Sets

A *Stripe Set* is a collection of *Step Ranges*.

A *Step Range* is an integer set with minimum element A and maximum element B, with some step S. The set includes all elements $A + n(S)$, for any n, where $A + n(S) < n$. Formally, we define $\text{STEP}(A, B, S) \coloneqq \{n(S) + A : n \in \mathbb{N}, n \leq (B - A)/S\}$, where $A, B \in \mathbb{Z}, A \leq B$ and $S \in \mathbb{Z}, S \geq 1, (B - A \bmod S) \equiv 0$.

This representation is the most general - it can express any arbitrary set of integers. But, the in-memory representation can be difficult to manage.

Consider a stripe set: $A = \text{STEP}(5, 13, 2) \cup \text{STEP}(20, 26, 3) = \{5, 7, 9, 11, 20, 23\}$, and a stripe set $B = \text{STEP}(0, 100, 10) = \{0, 10, 20, 30, 40, 50, 60, 70, 80, 90\}$.

How do we add these, in such a way that the result has as few step ranges as possible? At present we use one simple argorithm: For each combinationation of step ranges α, β, take the one with the fewest elements (say α, for this example). For every element a in α, create a new range $\text{STEP}(\min(\beta) + a, \max(\beta) + a, \text{step}(\beta))$. Issues quickly arise after a number of operations, so this representation should be avoided.

5.2 Small Sets

Small Sets is a 1 KiB uncompressed bit field, with an arbitrary offset. This is intended to be used for large enumerable values.

A *Small Set* may be used as the backing store for a keyboard scancodes like in SDL2′s SDL_keyboard.h, (n.d.).

5.3 Contiguous Ranges

Ideally, most ranges we perform arithmetic on should be continuous. Addition is trivial, and multiplication with a constant just creates a new Step Range.

5.4 Dynamic Represntation

The possible values of any scalar is kept as one of the above types, with a descriminator, this structure is called `DynSet`. The type checker can construct a new `DynSet` in 2 ways:

1. Using a single value, a, (e.g. synthesizing a literal). This creates a contiguous range from a to a.
2. Using a type range expression, `a..b`, this creates a contiguous range from `a` to `b`.

From the start of its life as a contiguous range, these dynamic sets can be *upgraded* to a more suitable representation. For example, after taking the union of two dynamic sets with no overlap, they'll be represented as a stripe set.

6 Constraints

At the type level, a boolean expression is considered an integer constraint satisifiability problem. The broad implications are discussed in Section 4.2.

Currently we support binary constraints involving multiplcation and addition. To find these constraints within the abstract syntax tree, we use a similar approach to type checking. Every node may or may not be represented as a *Constraint Term*. A constraint term may be a constant; variable; addition or multiplication between a variable and constant; addition or multiplcation between two variables; a variable constrainted to a set; or a constraint between two variables.

The process for mapping nodes to atomic constraint terms follows:

- Identifier referencing a mutable variable → *variable*
- Field access to a mutable struct field → *variable*
- Expression with a scalar result → *constant*
- Identifier referencing a immutable variable → *constant*
- Field access to a immutable struct field → *constant*

From these, we build compound terms:

- *variable* $+, \times$ *constant* → *unary operation*

- *constant* $+, \times$ *constant* $\rightarrow$ *constant*
- *variable* $+, \times$ *variable* $\rightarrow$ *binary operation*
- *variable* $<, >, \leq, \geq, \neq, =$ *variable* $\rightarrow$ *binary constraint*
- *variable* $<, >, \leq, \geq, \neq m =$ *constant* $\rightarrow$ *unary constraint*
- *variable* $<, >, \leq, \geq, \neq m =$ *unary operation* $\rightarrow$ *binary constraint*

A collection of unary and binary constraints, combined with the logical-and operator (`&&`) form a constraint set. We reduce each of the variable's values to satisfy the constraint, or, warn the user that the consideration will never be satisfied if this fails. Because we only handle expressions involving two mutable variables, expressions which do not meet this criteria are ignored.

6.1 Solving Constraints

We solve constraints with a naive constraint propogation algorithm, based on the algorithms described in *Foundations of Artificial Intelligence*, Chapter 3, Bessiere (2006). A constraint set is a collection of variables, and constraints on those variables. All variables begin with some unary constraint, by default this is that they must be a subset of their current synthesized type. From there, we iterate over every unsatisfied constraint, each constraint "propogates", returning that either it has been satisfied, it cannot be satisfied, or a mutation to some variable that is required for it to be satisfied (although its not guarenteed that it will be satisfied immatiately after the mutation is made). This scheme was originall inspired by Zhou's *Action Rules* language (Zhou (2006)). The search is depth-first, if a constraint is found to be unsatisfiable, we undo the last mutation and move up the tree. The first set of mutations that satisfy all constraints is used to produce the final collection of values.

7 Code Generation

Similar to type checking, code generation works by folding the abstract syntax tree. Each node writes to an buffer of assembly instructions, provided by the parent node. They return a collection of *Slots* which contain the value of their computation. A Slot may be a register, a pointer (itself a slot) and an associated offset, or a 16-bit immidiate. No optimizations are performed, and the generated code is generally ineficient, even compared to other compilers when they skip the optimization step.

References

Bessiere, C. (2006) Chapter 3 - Constraint Propagation Rossi, F., Beek, P. van and Walsh, T. (eds.). *Foundations of Artificial Intelligence*. 2 pp 29–83, doi: 10.1016/S1574-6526(06)80007-6

Dunfield, J. and Krishnaswami, N. (2020) *Bidirectional Typing*, doi: 10.48550/arXiv.1908.05839

Go (2024). https://github.com/golang/go

Hermans, F. (2024) *Hedy: A Gradual Language for Programming Education*, doi: https://dl.acm.org/doi/10.1145/3372782.3406262

Pike, R. (2012) *Go at Google: Language Design in the Service of Software Engineering - The Go Programming Language*, https://go.dev/talks/2012/splash.article

SDL2/Include/SDL_keyboard.h at Master · Zielmicha/SDL2 (n.d.). https://github.com/zielmicha/SDL2/blob/master/include/SDL_keyboard.h

Zhou, N.-F. (2006) Programming Finite-Domain Constraint Propagators in Action Rules, *Theory and Practice of Logic Programming*, 6(5), 483–507, doi: 10.1017/S1471068405002590