A Low Level Language with Precise Integer Types

Ian Shehadeh IRHSHEADEH@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 Introduction

Memory safety in systems programming languages has garnered a lot of attention in the last several years. A compiler that enforces strict rules on 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 guarentees.

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 Syntax

```
func boundedAdd(a: u32, b: u32): u32 {
  if U32_MAX - a > b {
    U32_MAX
  } else {
    a + b
  }
}
```

Listing 1: Addition without Overflow

Howlite's syntax prioritizes familiarity, ease of parsing, and clarity. The syntax should be familiar, someone unfamiliar with the language should be able to immediately grasp the programmer's intent, event if they do not understand every line. In a similar vein, the programmer should be guided towards writing code that is easily legible by others. We approach this issue by providing language constructs that clearly express intent. 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.

2.1 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.

2.2 Clarity

TODO

3 Type Checking

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 2: 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.

3.1 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 number are out of scope). A scalar may be any set of Integers.

3.1.1 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:

- $T \times U = \{1(-5), 2(-5), 3(-5), 1(-7), 2(-7), 3(-7))\} =$
  $\{-5, -10, -15, -7, -14, -21\}$
- $T + U = \{1 + -5, 2 + -5, 3 + -5, 1 + -7, 2 + -7, 3 + -7) =$
  $\{-4, -3, -2, -6, -5, -4\}$
- $T - U = \{1 - (-5), 2 - (-5), 3 - (-5), 1 - (-7), 2 - (-7), 3 - (-7)) =$
  $\{6, 7, 8, 9, 10\}$
- $T \div U = \{1 \div (-5), 2 \div (-5), 3 \div (-5), 1 \div (-7), 2 \div (-7), 3 \div (-7)) = \{0\}$

3.1.2 Bitwise Operations

Howlite times to do not explicitly define their size (in practice, all scalars are encoded as register-sized twos complement integers). This makes operations like bitwise not (set complement) difficult to define.

Consider the following `C` code, wich sets the n'th bit of a 16-bit field to zero:

```
uint16_t n = 9;
uint16_t field = 0b1010101010110011;
uint16_t mask = ~(1u << n);         // = 0b1111110111111111
field &= mask                       // = 0b1010100010110011
```

Translating to Howlite:

```
let n: 9 = 9;
let field: UInt16 = 0b1010_1010_1011_0011;
let mask: UInt16 = ~(1 << n);
field = field & mask;
```

Typechecking line three proceeds as follows, we begin from the bottom and synthesize up:

1. *synthesize* `1 : 1`, through the literal synthesis rule
2. *synthesize* `n : 9`, by using the variable's assumed type
3. *synthesize* `1 << n : 1(2⁹)`, using arithmetic synthesis rules

Now, we have the expression `(1 << n) : 0b1000000000`. Bitwise not of `0b1000000000` is `0b0111111111`.

Now, taking the intersection of `field` and `mask`:

$$\begin{aligned} \text{field}: &\ 1010101010110011 \\ \text{mask}: &\ \ \ \ \ \ 0111111111 \\ \& &\ \ \ \ \ \ 0010110011 \end{aligned} \tag{1}$$

Because `mask`'s synthesized type is only defined up to the 10′th bit, we accidentally clear the last 6 bits of `field`.

Listing 3: Example: Working with Masks

To compensate for the lack of explicitly sized integers, we define scalar *universe* types: `Pad8<{T : 0..0xff}>`, `Pad16<{T 0..0xffff}>`, `Pad32<{T : 0..0xffffffff}>`, `Pad64<{T :`

`0..0xffffffffffffffff}>`. We define equivalent types for signed integers: `Sign8<{T : -0x80..0x7F}>`, `Sign16<{T: -0x8000..0x7fff}>`, `Sign32<{T : -0x80000000..0x7fffffff}>`, `Sign64<{T : 0x8000000000000000..-0x7fffffffffffffff}>`.

References

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