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

This paper covers the process of designing a programming language. We present *Howlite* a language targeting RISC-V, intended for computation coupled to the underlying hardware. The language is a case study how structural typing and precise integer types can describe data structures at a low level.

Keywords: programming language

1 Programming Language Concepts and Motivation

Programming Language is a broad term, generally a programming language is a text-based format for expressing 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. Once a program is written it is read by both a machine (the compiler or interpreter) and humans. To satisfy both audiences, it must be clear in two ways: first, it needs to be unambiguous so the machine can produce consistent and accurate results; second, it must be expressive, meaning the author's intent should be apparent. To give context about what choices language designers might make, depending on their audience, we introduce *Hedy*, and *Go*.

1.1 Hedy

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

Example 1: Hedy (English)

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

Example 2: Hedy (Arabic)

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. The programs in both Example 1 and Example 2 print "Hello!" and ask the user their name, but their 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 they have a relatively narrow application.

In most programming languages, getting rich feedback from a program requires using complex graphics and audio libraries. To help new programmers see results quickly Hedy bundles graphics and audio output into the core language.

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, the problem this program solves and the algorithms used to solve it should be apparent. Go also comes bundled with tools to keep programs up to date and consistent. For example, unused variables and imports are disallowed. Although it supports first class functions, it's largely imperative.

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

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

  return -1;
}
```

Example 3: Go

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 guarantees.

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 that trade clarity for small guarantees.

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 specific data structure, wrapped in a Python library. It is a language for writing a bootloader or the entry point to a kernel. The compiler does not impose strict requirements on how the programmer manages memory or accesses data. Instead, the type systems provide a rich set of tools, that enable programmers to precisely describe how data is transformed.

2.1 Overview

```
type Boolean = 0 | 1;

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

Example 5: Boolean Type

The most notable feature of Howlite is the type system. The type system is structural and closely tracks the value of integers. For example, you can declare types that only allow the values `1`, `2`, or `5`. Types are compared based on their compatibility, not by name. For example, the data structure `{ x: int, y: int, z: int }` is compatible with the type `{ x: int, y: int }`. To better understand the language, this section will walk through the process of defining a function to get the index of a character from an ASCII string.

```
type char = 0..127;
type Uint32 = 0..0xffffffff;
type NatI32 = s32[0..0x7fffffff];
```

Example 6: Range Types

First, we define a character as any number between `0` and `127` (i.e. 7-bit ASCII characters). Next is the definition of a standard 32-bit integer, which is used to index the array. Finally, we define a variant of `i32` that is only positive. We'll use this type to represent the index, which can't be negative.

Now, we move on to the function signature. The syntax will be familiar to Go programmers, with a few small changes.

```
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 is referred to as `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 known as a fat pointer). A slice is simply a pointer and length pair; practically it functions like an array. Slice 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. 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 of a block is equal to the value of the last line in the block, if it omits a semi-colon (`;`), or `unit` otherwise.

```
{
  let i: Uint32 = 0;
  while i < str.len {
    if str[i] == c {
        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 answer is that 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 changed 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 should also clearly reflect exactly what the machine will do when executing the compiled program.

```
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

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 for functions. Instead we use the "let" or "func" keywords, respectively. These keywords and symbols were decided by picking from 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 is 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

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. Ultimately, 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.

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 unnecessary 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 convenient, they can paper over important details like memory allocation.

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 creating an unambiguous grammar, with no constructs that require unbounded look-ahead.g an unambigous grammar, with no constructs that require unbounded look-ahead.

4 Type Checking

Howlite 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: u32 = 1;
```

Example 10: Let Statement

Here, `u32` is the assumed type of `x`. Where ever `x` is referenced, we can consider it of type `u32`. 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.

4.1 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
}
```

Example 11: Average Three Numbers

The function parameters: `x`, `y`, and `z` are each given the assumed type `0..10`. An assumed type is analogous to the statement "no matter the value of `x`, we can always assume it is a `0..10`". The function's assumed return type is `0..10`. This allows any caller to treat the expression `average(a, b, c)` as a `0..10`, even if the operations performed by the function are unknown. An assumed type is a promise; it allows references to an 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.

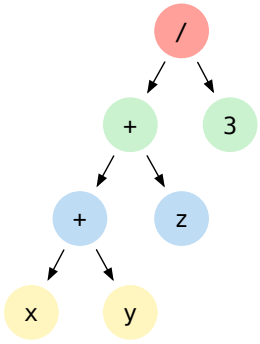
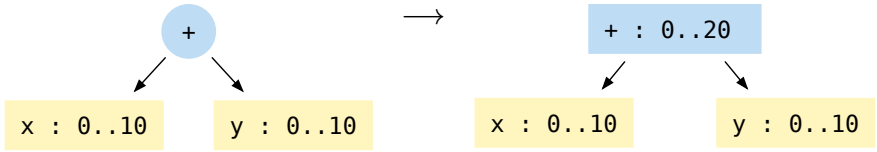


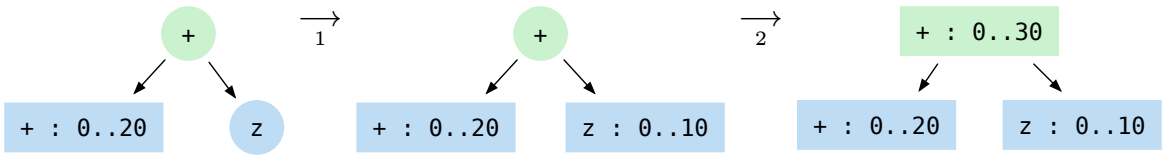
Figure 1: AST

The function 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 to 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 5. For now, take for granted that `0..10 + 0..10 : 0..20`.

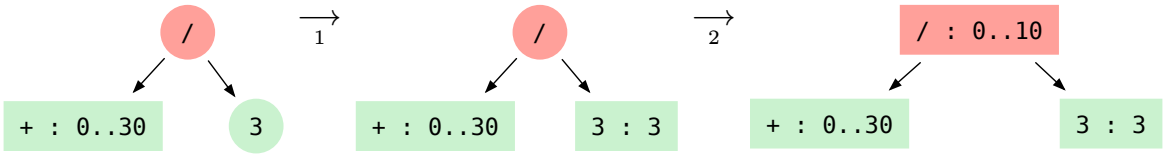


Now, we move up the tree, to synthesize the right-hand side of `+`, then finally `+` itself.



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

5 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). This collection of types contains any set of integers that can fit within a single general-purpose register on the target architecture.

Going forward, integer types will be expressed using the language's syntax: `1 | 3 | 5` is a type which can be constructed from any of the integers `1`, `3`, or `5`. The type `1..10` can be constructed from `1`, `10`, or any integer between the two.

5.1 Storage Classes

Scalar types belong to a *storage class* that identifies how they are encoded in memory. A storage class defines how many bits the scalar may use, and if one of them is a sign bit.

All integers are assumed to be two's complement. Consequently all integer overflow and underflow is well defined to wrap. For example, assuming all numbers in the following expression have a signed, 8-bit storage class, we find `-128 + -128 = 1`.

This mechanism plays well with our concept of scalar types: overflow is allowed, so it doesn't need to be policed if the programmer expects it. For example, consider a function which averages some set of numbers

```
func average(nums: &[Uint32; 1..1024]): Uint32 {
  let i: Uint32 = 0;
  let acc: Uint32 = 0;
  while i < nums.len {
    acc = acc + nums[i];
    i = i + 1;
  };

  acc / i
}
```

Example 12: Average an Arbirary Array

It's trivial to overflow cause overflow when adding `acc + nums[i]` (for example, if `nums = [0xffffffff, 0xffffffff]`). But if the author is concerned more with rapid development, or performance, they may not want to handle this case.

However, if overflow is known to be harmful then it can be explicitly forbidden. For example. Suppose we're reading a 64-bit ELF file, a common executable file format on Unix-like systems, we can read the address and size of a particular section from the 4th and 5th words of its *Section Heder* entry (2013):

```
let sh_offset: Uint64 = sh_entry[3];
let sh_size: Uint64 = sh_entry[4];
```

We know the file's headers take up at least 184 bytes, and Howlite enables this invariant to be encoded in the type system.

```
let sh_end: 0xB8..Max[Uint64] = sh_offset + sh_size;
```

This fails to compile, since `sh_offset + sh_size` might overflow, and wrap to a number less than both of them.

5.2 Construction of Scalars from Literals

A scalar is constructed by arithmetic operations, character literals or integer literals. We will use Howlite's own type construction syntax going forward: the given an expression `e` and a type `T`, the expression `e : T` asserts `e` constructs the type `T`.

A literal scalar can be constructed from a character literal, like `'A'`, `'\n'`, or `'🤯'`. The type constructed is a single value, equal to their Unicode codepoint. So, `'A' : 0x41`, `'\n' : 0xA`, `'🤯' : 0x1F92F`.

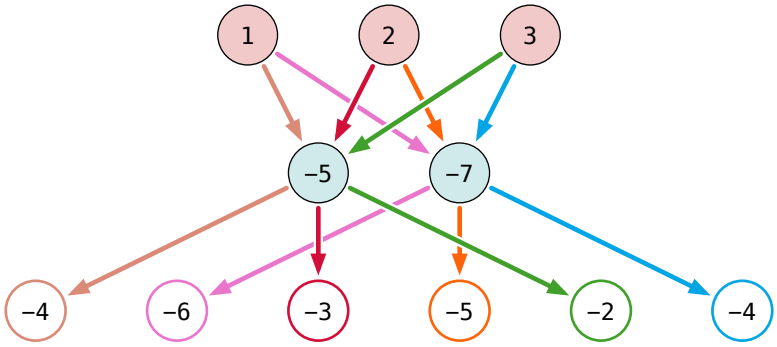
Literals can also be constructed from unsigned integers: `3 : 3`, `5 : 5`, `0b111 : 7`.

5.3 Construction of Scalars from Prefix Operators

The type-checker currently handles the prefix operators `!` (logical not) and `+`, and `-`. The `+` sign is a no-op, it's included in the language for cases where it might improve clarity. `-e` constructs the negative of `e`: it's equivalent to the expression `0 - e`. `!` has three cases: `!a : 0` if the type of `a` does not contain `0`, `!a` is `1` if the type of `a` only contains `0`, and `!a : 0 | 1` otherwise.

5.4 Construction of Scalars from Arithmetic Operators

Addition and subtraction operators (`+` and `-`) produce a type representing every possible sum of the operands' types, and no more. For example, given a variable `a: 1..3`, and a variable `b: -5 | -7`, then the expression `a + b` has the type `-6..-2`.



For performance, multiplication and division produce only a contiguous range from the minimum possible result to the maximum. So, re-using the variables defined above, we find `a * b` has the type `-21..-5`, even if the expression can only produce $-21, -15, -14, -10, -7,$ and -5.

5.5 Construction of Scalars from Comparison & Logical Operators

Unconditionally, all comparison operators: `<`, `<=`, `>`, `>=`, `==`, and `!=` synthesize to the type `0 | 1`. Similarly, logical "and" (`&&`), and logical "or" (`||`) always synthesizes to the type `0 | 1`.

Although a scheme similar to the logical not (`!`) operator could be implemented, where the constructed type depends on the operands, a simple implementation was chosen for development efficiency, and to see how it would effect programming in the language. Ultimately, the difference rarely matters, since if the outcome of a boolean operation is always true, or always false its likely either for debugging, or an error on the programmer's part, making the case relatively rare.

5.6 Future Work

The largest missing piece are bit-wise operations. Due to the internal representation of integer sets (discussed in Section 6), it is difficult to compute bit-wise "XOR", "AND", and "OR" operations.

6 Disjoint Integer Sets

Integer sets are used throughout the type checker. As described in Section 5, these sets must be able to every possible result of addition, subtraction, and *at least* determine the upper and lower bounds of other operations. Further, the type checker will often test subset relations between sets, and union a series of sets.

There are many solutions for storing large disjoint sets of integers: in particular we investigated Roaring Bitmaps (Chambi et al (2016)), and Range Set Blaze (Kadie (2024)). The set representation was developed with the intention of tracking the exact results of multiplication, and division not just addition and subtraction. To this end, we chose using

a list of stepped ranges, instead of continuous ranges like Range Set Blaze. However, this representation made simple operations, like subset difficult, so, to optimize cases where we have sets of arbitrary values, we also give the option of using a large uncompressed bit map. Finally, to optimize the typical case, where the programmer is performing arithmetic on a large continuous range, sets may be represented just using the two endpoints.

The following sections give a more detailed overview of the three representations.

6.1 Stripe Sets

A *Stripe Set* is a collection of *step ranges*. A step range is some set which includes a minimum element A, a maximum element B, and every N^{th} element between the two. For example, we could have a range with a step of 5, from 0 to 15, which includes 0, 5, 10, 15

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 $S \in \mathbb{Z}, S \geq 1, (B - A \bmod S) \equiv 0$.

In order to add two step ranges: α, and β we take the one with the fewest elements (say α, for this example). For every element a in α, create a new range $\text{STEP}(\min(\beta) + a, \max(\beta) + a, \text{step}(\beta))$. Multiplication and division falls to only operating on the set's minimum and maximum, in order to construct a new continuous set.

This representation is the most general - it can express any arbitrary set of integers. But, the in-memory representation can be difficult to manage. Operations like union and set subtraction can cause the internal representation to become fragmented - several step ranges are used to express a collection of values that could be expressed with a single step range. Care is taken to avoid this fragmentation, or correct it when detected, however the algorithm is far from perfect. During development we found taking union of roughly 60 continuous ranges had a large decrease in performance.

6.2 Small Sets

A small set is a 1KiB array of bits, with an arbitrary offset. A value N is included in the set if the bit at index $(N - \text{offset})$ is set.

This was originally conceived as a way to help with bit-wise operations. Ideally, the small size, but quick set operations would be a good trade-off for representing enumerable types or bit flags. However, it is still difficult to compute every possible result of a particular bit-wise operation between two small sets, making them unfit for this use case. Since enumerable types and bit-flags typically only have a relatively small set of defined values, it would likely be more efficient to use an array of integers.

In the current iteration of the type checker, small sets are often used to store the type of string literals. String literals are short-hand syntax for an array of UTF-8 encoded characters. Usually, the element type for these arrays contains several arbitrary integers, all clustered between 0 and 255 (for example: the string "Hello World" has the type [32 | 72 | 85 | 100 | 101 | 108 | 117 | 120; 11])

6.3 Continuous Ranges

A continuous range is a set with a minimum element A, and a maximum element B, which includes every integer between the two. This is the ideal representation, since addition

and subtraction can be easily computed. To precisely compute the possible results of multiplication between two arbitrary ranges is more complicated, but as mentioned above we only get the smallest possible continuous range for multiplication, making the operation relatively cheap.

6.4 Dynamic Representation

The possible values of any scalar are kept as one of the above types, with a discriminator, 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. In the current implementation, sets will never be *downgraded*, although it is theoretically possible. This can have odd effects on performance, since a simple set of integers may have an overly complex representation. For example, if we add `0..5` (a continuous range) to the set `0 | 5 | 10 | 15` (a stripe set), the result would be `0..20`, represented as a stripe set.

7 Type Narrowing

In addition to flow control, conditionals also allow the programmer to narrow types. If x is some unsigned number, a statement like `if x < 5 { ... }` will give x the type 0..5 within the body of the if statement. If the conditional has an attached `else` clause, then the inverse of the condition is used to narrow types within the body of the else clause. So, the statement `if x < 5 { ... } else { ... }` is semantically equivalent to `if x < 5 { ... }`, followed by `if x >= 5 { ... }`.

In practice, the "narrowed type" of a variable is a third layer on top of the existing assumed and synthesized types. Each scope may define a single narrowed type for a variable, which overrides the synthesized type in that scope alone. Narrowed types are distinct from synthesized types because they are associated with a scope: they have no effect on the variable's type outside of that scope, and only the first narrowed type in the scope heirarchy matters. For example, consider a snippet which narrows the value of x twice:

```
if x < 10 {
  // x has a narrowed type of 0..10
  if x < 5 {
    // x has a narrowed type of 0..5
  }
  // x has a narrowed type of 0..10
}
```

If we instead assigned x to 5 within the if-statement inner, this would effect its synthesized type, which would be kept past the end of the if-statmeents body. In short: Assumed types are axioms, synthesized types observations based on assignment, and narrowed types are observations based on conditions.

7.1 Loops

Currently, only while loops are supported. For simplicity, since while loops can run an indefinite number of times they are treated as a black box. When a while loop is encountered, all variables are assigned their assumed type. Meaning, an previous range analysis is ignored.

```
let a : 0..10 = 3;
while a < 5 { a = a + 1; }
let b: 0..5 = a; // type error - a is treated as a 0..10
```

Example 13: While Loop Narrowing

Within the body of the while loop, types are narrowed using the while loop's condition. In Example 13, line 2 we can safely increment a, since the condition ensures $a : 0..4$, but the assumed type of a is 0..10.

7.2 Producing Constraints

Constraints are constructed from the condition of if-statements and while loops, by a *model builder*. Similar to type checking, the model builder reduces the syntax tree bottom-up. Each node is converted into a *term*. A term has three possible shapes:

1. An *Atom*: a variable with an offset: for example "x + 3", "y - 10".
2. A *Cond*: a list of clauses, which must all be true
3. A *LinearSum*: some expression in the form $ax + by + ... + c$, where a, b, c are constant, and x, y are variables.

If an expression cannot be fit in one of the above forms, then the model builder ignores that expression. For example, if x, and y are variables, then $x * y$ cannot be expressed in any of the above forms, so this expression would be ignored. Ignored expressions cascade: if the full expression was $(x * y) + 100$, then the model checker would throw away everything, since it cannot reason about $x * y$. However, every condition separated by the `&&` operator is considered separately.

The builder stops at comparison operations and adds it to the list of possible clauses in the model, producing a *Cond* term.

The aries library [Bit-Monnot (2024)] is used for solving constraints. Because the library only supported 32-bit signed integers, and the Howlite type-checker relies on 128-bit signed integers, we maintain a fork of the library with support for 64-bit, and 128-bit integers. Ideally these changes can be added to upstream repository.

7.3 Solving Constraints

While adding constraints, the model builder maintains a list of variables, or data structure fields that may be narrowed. Once the aries solver runs, the type checker repeatedly queres the domain of these variables. After each query, the returned range is added to the variable's new domain, then the solver's model is updated to eliminate that range.

```
# begin with an empty assignment for each variable
var_type = IntegerType.empty()

# search for a solution by repeated backtracking all invalid decisions
# then propogating constraints.
while solver.backtrace_and_propogate():
  for var in variables:
    var_domain = solver.get_domain(var)
    var_type.include_range(var_domain.lower_bound(), var_domain.upper_bound())
```

Figure 1: Repeatedly Query the Solver to find all possible solutions

Once we have a set of possible assignments for each variable, the solutions are integrated back into the type checker. This process is similar to assignment. If an entire variable is narrowed, e.g. $a < 30$, then that variable in assigned a new narrowed type equal to the solution found by the solver, within the relevant scope.

If only a single field of a variable is narrowed, for example `err.kind == 1`, then we copy the variable's type, replace the field with the new type produced by the solver, then process any consequences of that. For example, if the variable's type is a union, and that assignment is illegal in some of the variants, then those variants are thrown away. A possible use-case for this can be seen in the following example of a function to print compiler errors.

```
type FsError = { kind: 0, filename: Str, errno: Int32 };
type ParseError = { kind: 1, filename: Str, token: Str, line: UInt32 };
type LexError = { kind: 2, filename: Str, char: Str, offset: UInt32 };
type Error = FsError | ParseError | LexError;

func print_error(err: Error): unit {
    print(&"Error in ");
    print(err.filename);
    if err.kind == 1 {
        print(&" parse error at line ");
        print(int2str(err.line));
    };
    print(&"\n");
}
```

Example 14: Howlite

Notice we can access both the `err.kind` and `err.filename` fields without narrowing, since those exist on each union variant. But, in order to get the line number for parse error the variable has to be narrowed by testing the value of `err.kind`.

8 Conculsion

References

Bit-Monnot, A. (2024) *Plaans/Aries: Toolbox for Automated Planning and Combinatorial Solving.*

Chambi, S., Lemire, D., Kaser, O., et al (2016) Better Bitmap Performance with Roaring Bitmaps, *Software: Practice and Experience*, 46(5), 709–719, doi: 10.1002/spe.2325

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

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

Kadie, C. (2024) *CarlKCarlK/Range-Set-Blaze*

Pike, R. (2012) *Go at Google: Language Design in the Service of Software Engineering - The Go Programming Language*

System V Application Binary Interface, Chapter 4, Sections (2013).