Hobbes

Feb 19, 2021

Introduction

Hobbes is a domain-targeting programming language and execution environment built and maintained at Morgan Stanley.

By design, it fulfills three major goals:

- Dynamic, in-process rewriting of processing rules for domain objects (trades, orders, executions)
- Persistence and out-of-band processing of structured logs for order managers and surrounding processes
- Rock solid, ultra-low latency execution

Hobbes was developed to manage the runtime of low-latency processes such as equities trading engines, which generally cannot be restarted during the working day.

The target user base is the development and production management teams responsible for building and maintaining these processes in production. As such, Hobbes is *obsessively* pragmatic: the vast majority of design choices are aimed at fulfilling these needs.

Perhaps most surprisingly, Hobbes is a variant of the pure-functional programming language *Haskell*. The following is an example of some Hobbes code from a production system at Morgan Stanley:

```
nil :: () -> (^x.(()+(a*x)))
nil _ = roll(|0=()|)

cons :: (a, ^x.(()+(a*x))) -> (^x.(()+(a*x)))
cons x xs = roll(|1=(x,xs)|)
```

Read on to discover more about the Hobbes language - its design and purpose, and how you can use it in your systems!

Note: Hobbes Usage

Hobbes is built for high performance integration with C/C++ applications. While Hobbes is a strongly typed language that offers compile-time checks, it doesn't have a sandboxed runtime environment or runtime safety features. By design, Hobbes gives direct access to memory and does not have array bounds checks. Additionally, Hobbes supports compilation and execution of native code remotely over a network (RPC). This feature is meant for use within your trusted internal network only. If you choose to utilize such functionality, you need to be aware of these design choices and understand the security implications.

CHAPTER 1

Contribution

A step by step guide on how to contribute to Hobbes can be found *here*.

CHAPTER 2

License

License information here.

2.1 Components

Hobbes is comprised of two main components:

2.1.1 A programming language

Hobbes is a haskell-like programming language with a rich type system. Hobbes code can be embedded in a C++ program and data marshalled between the two. This means you can write C++ code which binds against a Hobbes environment and calls into Hobbes functions to execute functionality.

Over the day and as business requirements change, you can update the bound Hobbes code to give it different behaviour. The new code is compiled to highly efficient x86 instructions for later execution.

Similarly, you can make C++ functions available to the embedded Hobbes code. Ultimately, this gives you the power to choose the most efficient and effective format for different parts of your codebase: Highly structured and well-defined parts of your application can be built using C++, whilst the dynamic business logic can be written in Hobbes.

For more details about hosting Hobbes in a C++ application, see *Embedding Hobbes*.

2.1.2 A persistence format

Secondly, Hobbes comes with a typesafe, space-efficient persistence format for realtime storage and retrieval of application data.

This can be used for inter-process communication over TCP, quering and filtering of daily application logs, or fast post-hoc analysis of application behaviour based on Hobbes' internal decision tree structure.

For more details about Hobbes' persistence format, see *Structured logging in Hobbes*.

Rationale

Why create a custom programming language and persistence format? For a deeper look into the minds of its developers, you'll want to read *Digging into the domain*

2.2 Digging into the domain

Taking a deeper look into our domain might help clear things up.

Hobbes has been designed from the ground up, specifically to help devops staff manage the in-process configuration of extremely low latency Order Managers. An Order Manager (sometimes just called the "OM") is a key component in a trading system - it's responsible for maintaining the state of all the trade orders the organisation has received and is processing.

The OM takes instructions in the form "Buy 100 stocks of *Example co.* if the price is below 144.2 USD". This is called a *limit order*, because the action is contingent on some property of the stock (in this case, its price).

The Order Manager will hold this order and watch the price at which *Example co.* is being sold at different exchanges, and once the price drops below 144.2 it'll go ahead and issue an instruction to buy the stock. Some time later the exchange will respond with the result: either the purchase succeeded and the order filled, or else the details of the failure.

The alternative to a *limit order* is a *market order*, where the stock is bought regardless of the current price.

2.2.1 Complex Business

There's some added complexity, though. One issue is that trades are visible to other players in the market, and if we execute a large order all at once, we'll affect the price of the stock to our detriment. A number of trading strategies exist to constrain the effects of an order on the market - this is the basis of *algo trading*.

Further, a client may make further conditions on the execution of an order: they may want to split the order across a number of trading venues (half to NYSE, half to LSE). Depending on the client, you may wish to offer them credit. There may be long-standing *master agreements* between parties that form the basis of a clients trading decisions, that the OM must take into account when deciding when, where, and how to place orders.

This complexity mounts very quickly, and must be managed dynamically – in both senses: *quickly* and *at runtime*!

However, the go-to tools which developers use to manage complex decision trees full of runtime information (type hierarchies with virtual function calls; long if-else chains or switches) are wildly inappropriate in the hot path of a low-latency OM.

Two things are important:

- The ability to construct and compile a decision tree for a given stock
- The ability to quickly change the behaviour when given new constraints

Both the dynamic portion (the operands to function calls and control flow statements; e.g. the stock's price) *and* the static portion (the operations themselves) must be changeable.

In particular: in-process, compiled logic allows the processor to maximise efficiency by filling an instruction and data pipeline, thus enjoying the benefits of mechanical sympathy. The output of the Hobbes compiler lives in a critical trading path where it sees a very large proportion of daily US equities trades.

Note: Why a custom language?

It's a great question. The domain (dynamic, non-developer-driven changes to execution behaviour of processes which can't be restarted, and which come with a *tight* latency budget) is specialised enough to quickly exhaust most existing solutions:

- Hosting a python environment would be user-friendly but not fast enough.
- Dynamically-compiled C++ would be fast but ugly, brittle, and complex.
- Marshalling runtime execution decisions to another process, perhaps one which enjoyed more natural dynamic mapping to natural language, would quickly blow latency away.

In addition, a custom *type system* allows for tight bindings with existing data in hosting applications, along with low-latency custom serialisation for binary logging.

2.2.2 Why not Haskell?

Despite some similarities, the Hobbes programming language is emphatically *not* a Haskell clone. There are some major differences:

- Hobbes is not a pure functional language
- Execution of Hobbes code is *strict*, not lazy. Data dependencies are evaluated when they're defined, not when they're used.
- There's no implementation of the Haskell *boot* libraries - the core standard library that exists by contract in a Haskell implementation.
- There's no implementation of Haskell *core* - the typechecked, mostly-unchanging barebones language that all Haskell language extensions ultimately compile down to.

2.3 The Type System

Like many functional-style programming languages, the power of Hobbes lies in its rich type system, so that's where we'll start. There are primitive types, arrays, record types, tuples, and variants.

Hint: hi, the Hobbes interpreter

You can execute most of the code examples we'll show here in hi, the interpreter that comes packaged with Hobbes. For more information, take a look at the chapter *hi, the Hobbes interpreter*

2.3.1 Primitive Types

The Hobbes primitive types are arranged in memory in a manner which allows for free marshalling when Hobbes is executing in a C++ process. Each of the primitive types has a simple literal syntax allowing for the easy initialisation of a value:

Unit A simple null-type with only one value

```
> ()
```

Bool Either `true` or `false`

```
> true
true
```

Char A single character of text

```
> 's'
s
```

Byte A single byte (`0-255`)

```
> 0Xad
0Xad
```

Short A two-byte number

```
> 3S
3S
```

Int A four-byte number

```
> 4
4
```

Long An eight-byte number

```
> 5L
5
```

Float A single-precision (4 byte) floating-point number

```
> 3.0f
3f
```

Note: float literals must be a decimal number followed by the character 'f'.

Double A double-precision (8-byte) floating-point number

```
> 3.0
3
```

2.3.2 Arrays

Just like in C++, Hobbes arrays are zero-indexed, contiguous values in memory, with special syntax for `char` and `byte` arrays. Hobbes supports bounds-checking to prevent a common class of bug by maintaining the array length alongside the data in memory.

```
> [0, 1, 1, 2, 3]
[0, 1, 1, 2, 3]

> "Hello, Hobbes!"
"Hello, Hobbes!"

> 0xdeadbeef
0xdeadbeef
```

Note: Strings

In Hobbes, a String is simply an array of `char`.

Warning: 0x versus 0X

It's important the note the subtle difference between the literal syntax for `byte` and for `byte` *arrays* - the case of the 'X' is very important!

Uppercase for `byte`, lowercase for `byte` array.

Array functions

A number of functions are overloaded for array types:

```
> [0, 1, 2] ++ [3, 4, 5]
[0, 1, 2, 3, 4, 5]

> size([0,1,2])
3
```

You can index into an array using square brackets:

```
> nums = [6, 2, 4, 6, 5, 9, 8, 5, 6, 3]
> nums[3]
6
```

In addition, the open and closed slice syntax is available:

```
> nums[3:6]
[6, 5, 9]
> nums[2:]
[4, 6, 5, 9, 8, 5, 6, 3]
```

You can read `[2:]` as "the second index, until the end". The converse works, too:

```
> nums[:2]
[3, 6, 5, 8, 9, 5, 6, 4]
```

This is "the end until the second index". Indexes from the end of the array can be counted with a *unary negate*:

```
> nums[2:-3]
[4, 6, 5, 9, 8]
```

And of course you do that in both positions in the slice:

```
> nums[-2:-4]
[5, 8]
```

Warning: Array indexes

Array indexes in Hobbes aren't bounds checked, so whilst you can *slice* from the end of an array, you can't use the same syntax to *index*:

```
> nums[-3]
51627831
```

In addition, attempts to slice off the end of an array will act as though you were slicing from the beginning or the end, respectively:

```
> nums[-2014:305]
[6, 2, 4, 6, 5, 9, 8, 5, 6, 3]
```

Array sequences

A *sequence expression* can be used to initialise an array of ints. The syntax is simple:

```
> [1..4]
[1, 2, 3, 4]
```

We can take this further and generate *infinite* sequences by leaving the upper bound open:

```
> [0..]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]...
```

Infinite sequences are useful structures for performing work many times over without mutating a loop variable. Special care has been taken to ensure their evaluation isn't eager, however - as your program might never stop! For more information about the *type* of an infinite sequence, take a look at the infinite squences section in *polymorphism*

2.3.3 Records

Records are a common way to keep closely-associated pieces of data together in functional progamming, and they're often referred to as an *and* type: a hostport is a host *and* a port - and that's it. No behaviour, and its identity is simply the two elements.

Record types are similar in spirit to C++ structs, with ad-hoc declaration and initialisation, plus type inference:

```
> {name="Sam", age=23, job="writer"}
```

Records are examples of structural types, meaning that in Hobbes, even though they are both examples of different anonymous ad-hoc types, the two are *equivalent*:

```
> {name="Sam", job="Writer"} == {job="Writer", name="Sam"}
true
```

Note: Equivalence vs Equality Although it's true to say that, in Hobbes, the two record instances above are *equivalent*, they're not *equal*, and so the following equality test would fail to compile:

```
> {name="Sam", job="Writer"} === {job="Writer", name="Sam"}
stdin:1,28-30: Cannot unify types: { name:[char], job:[char] } != { job:[char],␣
↪name:[char] }
```

This is because the equivalence relationship is determined not by any special logic in the Hobbes compiler, but by the equivalency type class `Equiv`. This class contains the implementation of == and thus decides how to unpack the record instances and compare them.

A type class is a way of describing expected behaviour on a type. In the Hi REPL, I can unpack the `Equiv` typeclass with `:c`:

```
> :c Equiv
class Equiv where
  == :: (#0 * #1) -> bool
```

For more information about typeclasses in Hobbes, see *Type Classes*.

2.3.4 Tuples

Like records but with no field names, tuples are used to keep commonly-associated data together. The canonical example is the host/port pair:

```
> endpoint = ("lndev1", 3923)
> endpoint
("lndev1", 3923)
```

Note: Assignment

Notice here that we've assigned the tuple to the name *endpoint*. This name now exists in the global context, which means you can refer to those values using the name "endpoint" from now on. For local scoping, see *Local scoping*

Note: Pretty-printing

Hobbes has good support for printing the primitive and scalar types: char arrays are printed as strings, the literal syntax is displayed when printing to *standard out*, etc.

When we deal with arrays of records or tuples, Hobbes gives us a convenient table notation:

```
> [{First=1, Second="two"},{First=3, Second="Four"},{First=5, Second="Six"}]

First Second
----- ------
    1    two
    3   Four
    5    Six
```

2.3.5 Variants

The variant is the richest way to declare a type in the Hobbes type system, because it gives us the opportunity to declare a value which can be one of a number of named cases. If the Record type is an *and*, the Variant is an *or*.

This allows us to model enum-like structures with associated data. In the following example, we're declaring a type called `status` which can model the success or failure of a service call. In the case of a failure, we'll be given an error code which we'll want to react to. However, in the successful case, there's nothing more to do:

```
type status = |
  Succeeded,
  Failed: int
|
```

(continues on next page)

(continued from previous page)

```
> status = |Succeeded| :: status
|Succeeded|
```

Warning: type declarations in hi

Hi doesn't currently support some Hobbes expressions, including type declarations. You can write your types in a file and have them loaded into a hi session by following the instructions in *Hi can load files*.

Note: Type Annotations

Sometimes Hobbes requires us to specify the type of a value. In the case above, we want to be careful about the instantiation of the `|Succeeded|` type: we need to be clear that we're instantiating a subtype of `status`, rather than a naked record type with just one subtype which happens to be called 'Succeeded'. hi can show us the inferred type of a value with `:t`:

```
> :t |Succeeded|
|Succeeded=()|::a=>a

> :t |Succeeded| :: status
|Succeeded, Failed:int|
```

The `::` allows us to specify the type of the variable using what's called a *type annotation*. More information about types and type annotations is available in *Polymorphism in Hobbes*.

As we'll see in *pattern matching*, Hobbes has rich language support for building logic based on variant types.

2.3.6 Sum types

Just as the tuple type can be thought of as simply a record using numbered placement instead of names, the sum is a variant without names: a true union.

```
> |1="hello"| :: (int+[char])
|1="hello"|
> |0=3| :: (int+[char])
|0=3|
```

In this case we're using the index (0 or 1) to specify the actual variant type we're using - int or char array. An instance of the first type must hold an int, and an instance of the second type must hold a char array - in this case, a String.

2.3.7 Recursive type definitions

Recursive types (i.e. a type whos definition includes itself) are a powerful way to define complex data structures and are commonly found in functional programming.

The canonical example is the list, which is defined as two parts: head and tail. The head is a single list element, whilst the tail is… another list!

In Hobbes, we declare that a type is recursive by simply giving it a name and denoting the name with a caret:

```
^x.(()+([char]*x))
```

In this type expression we use the caret to give a name to the type so it can be used recursively throughout the expression. In this case the list type, `x`, is declared as a sum type of an empty list, or a string and a list.

We can easily construct one using Hobbes's constructor syntax:

```
> cons(1, cons(2, cons(3, nil())))
1:2:3:[]
```

Whilst this construction syntax might look unwieldy, the generation of such structures is commonly algorhithmic, and (as discussed earlier), the payoff is in Hobbes's rich matching syntax.

Many structures in Hobbes can be defined recursively because, as we'll see, recursion is a deeply powerful element of functional programming.

2.4 Control Flow

The Hobbes language itself is quite small. Function definitions, as well as types, type classes, and their instances, must be written in a Hobbes file in order to be defined. These examples are therefore shown without the hi prompt - for more examples, see *hi can load files*

The control flow rules for Hobbes are similar to Haskell:

2.4.1 Functions

Functions are first declared by their type, and then implemented for specific values.

```
addOne :: int -> int
addOne s = s + 1
```

If we didn't first specify the type of the function, Hobbes would attempt to create a polymorphic function with parameter types restrcited simply by the way in which values of that type are used. For more about this behaviour, see *polymorphism*.

2.4.2 If/else

```
if a < 1 then 2 else 3
```

Note: Note that 'if' is an expression, meaning that it resolves to a value:

```
> b = if true then 12 else 13
> b
12
```

2.4.3 Pattern matching

Match expressions are perhaps the most powerful element of Hobbes syntax, because they let you perform actions based on the value or even type of other expressions.

In the simplest form, they're a bit like a switch statement:

```
match 3 with
| 1 -> show("hello")
| 2 -> show("hobbes")
| _ -> show("oops!")
```

Note: **underscores** In Hobbes match expressions, the underscore serves two purposes.

Firstly, it's a wildcard. In the above code the underscore is the default case that will be executed if all the above cases have failed to match.

Secondly, it's an instruction not to bind a name to the matched element. We'll see more about binding below!

When used with more than one value they can be used to match against any element in the set. However, the Hobbes compiler is smart, and will complain if you haven't provided a case for all the potential options. In the next example, we'll be caught red-handed with a so-called "inexhaustive match":

```
match 1 2 with
| 2 _ -> show("first!")
| 1 2 -> show("second!")
```

Warning: Inexhaustive patterns in match expression

We can always resolve this by adding a default case, or by providing cases for all the possible options - although this might mean writing a lot of code!

```
match 1 2 with
| 2 _ -> show("first!")
| 1 2 -> show("second!")
| _ _ -> show("default!")
```

In the above example, it's important to note that the matching works top-down, meaning that the first valid case will be evaluated:

```
match 1 2 with
| _ 2 -> show("matched!")
| 1 2 -> show("didn't match!")
| _ _ -> show("didn't even get here!")
```

The way to read the first case is "*any value* followed by the integer `2`". Even though the second match is more specific (*i.e.*, both elements match the values), it's the first case that's matched.

Also, note that because we're matching against two values, we have to use two underscores in the final case. If we fail to do that, Hobbes will tell us "row #3 has 1 columns, but should have 2".

Note: Just as `if` expressions can be written on one line, we can save space (and be more idiomatic) in our Hobbes code in the same way. The above match can be re-written as

```
match 1 2 with | _ 2 -> show("matched!") | 1 2 -> show("didn't match!") | _ _ -> show(
↪"didn't even get here!")
```

The Hobbes standard library is full of code like this, and Hobbes developers quickly get used to writing code this terse. You can decide what works best for you!

Matching and binding

As well as matching on values, we can also bind values to names within a match case. In the following example, we're matching on the first element of the tuple and binding to the second:

```
match 'a' 123 with
| 'a' fst -> show(fst)
| 'b' snd -> show(snd)
| _ _ -> show("default")
```

In each case, we're simply matching on the (char) value of the first element. If that matches, we bind the second element to a value. In the first case (which ultimately is matched), the name we give the value is `fst`, but there's nothing special about that; we could have called it anything. The name `fst` is then lexically scoped to the match expression following the arrow - it's not available in other cases, or outside the match.

Note: To some programmers, this "match and bind" behaviour seems strange, and it's another good example of the "terse vs powerful" battle often found in the minds of new functional programmers!

Tuples

Hobbes also lets us match against the values of tuple elements, leading to another common idiom. The ease with which we can match and bind using the match syntax with tuples means that ad-hoc tuples are often created simply to limit pollution of the global namespace with values which could be scoped more appropriately. Consider the below case:

```
match env getHostPort(env) with
| "dev" (host, port) -> connect(host, port)
| "qa" (host, port) -> connectqa(host, port, qadb)
| "prod" (host, _) -> connectkrb(host)
| _ _ -> ...
```

In this case we're creating a tuple simply for the purposes of immediately matching against its values and unpacking it.

Here again the underscore is used as a wildcard - in this case you can read it to mean "there *is* a value here but I don't care what it is, and I don't want to use it so don't even give it a name".

This matching-and-binding logic can be generalised to arrays, too:

```
match [("sam", 2013), ("james", 2012), ("stephen", 2010)] with
| [_ (n, 2012), _] -> show(n)
| _ -> show("none")
```

And, because of the way character arrays are matched, even to regular expressions:

```
match hostname with
| '.*qa$' -> show("qa")
| _ -> show("prod")
```

Guard matching

We can also match based on ranges of values, using a so-called "guard":

```
match 1 with
| x where x < 10 -> show("small!")
| _ -> show("large!")
```

Note: The rules for match expressions are simple: every case in the expression must be **reachable** (i.e., no previous row can have matched against all the possible values for this row) and the match table must be **exhaustive** (i.e. all possible cases must be matched against).

These rules combined explain why you so commonly see wildcard matches at the end of a match expression - the wildcard catches any cases that haven't previously been matched; and putting it at the end it prevents further cases from being unreachable.

Remember, the rule is *first possible match*, not *most specific match*!

Matching on Variants

Just like with tuples we can match on - and unpack - sum and variant types. Recall our status type from earlier:

```
type status = | Succeess, Failure: int|
```

We can write a matching function which classifies values of this type and acts accordingly:

```
classify :: status -> [char]
classify s = match s with
| |Success| -> "finished"
| |Failure=x| -> "failed with error" ++ show(x)
```

Similarly to the complex match expressions above, we can match on values as well, to provide special functionality for specific cases:

```
classify :: status -> [char]
classify s = match s with
| |Success| -> "Succeeded"
| |Failure=404| -> "Not Found"
| |Failure=err| -> "Error: " ++ show(err)
```

Match expressions

In the previous examples, we've been calling the unit `show` function in our match cases. But in Hobbes, just like with `if`, `match` is an expression - that means it's results can be assigned directly to a name:

```
hostport = match env with | "prod" -> "lnprd" | "qa" -> "euqa" | _ -> "ln123dev"
```

And just like with `if`, the *types* of the expressions in each branch must also match. Failing to ensure the cases are of the same type will result in an error:

```
> match 1 with | 0 -> "hello" | _ -> 1
Cannot unify types: [char] != int
```

Sugar

The Hobbes compiler uses match expressions “under the covers” in a number of different situations. For example, the `matches` keyword can be used to perform all the unpacking and pattern-matching that a single-case match statement can:

```
(1, 2) matches (1, 2)
```

is re-written by the Hobbes compiler to

```
match (1, 2) with
| (1, 2) -> true
| _ -> false
```

Similarly, these two are equivalent in Hobbes:

```
"sam" matches '..m'

match "sam" with
| '..m' -> true
| _ -> false
```

This process of conversion to another program structure is commonly called “desugaring”, because the nicer, lighter-weight style is known as “syntactic sugar”. There are many examples of sugaring in the Hobbes language, and we’ll try to point them out as we go. Sweet!

2.4.4 Tuple Decomposition

A tuple can be decomposed into its individual parts very simply:

```
> (host, port) = getHostPort("dev")
> host
"lndev01"
> :t port
int
```

2.4.5 Comprehensions

Similar to comprehensions in Python, these allow us to describe the algorithm used to create a sequence of data.

Note: Remember, Hobbes code is executed eagerly, meaning the comprehension will usually be evaluated in full when it is declared.

```
[show(x) | x <- [0..20], x % 3 == 0]
```

This can be read as “for each x in 0 to 20, where x is divisible by 3, show x”.

The comprehension is split into a mapping function, a generator expression, and a filter. The mapping function is applied to the results of the generator function where the filter holds true.

Note: sequence expressions

Look closely and you'll see the *sequence expression* syntax described in the types chapter. You could also use an inline array declaration here:

```
> [show(x) | x <- [11, 12, 13], x % 3 == 0]
```

Or alternatively the name of an array declared elsewhere:

```
> nums = [98, 99, 100]
> [show(x) | x <- nums, x % 3 == 0]
```

The comprehension syntax is an expression, and can therefore be used anywhere a range of elements is expected. For an example, the Hobbes standard library contains the following code:

```
productWith :: ((a, b) -> c, [a], [b]) -> [c]
productWith f xs ys = concat([[f(x,y) | y <- ys] | x <- xs])
```

This describes a function `productWith`, which combines the cross product of elements from two lists with a function:

```
> productWith((\x y.x+y),[1,2,3],[4,5,6])
[5, 6, 7, 6, 7, 8, 7, 8, 9]
```

If we were to write out this `productWith` function in pseudocode, it might look like this:

```
for(x in [1, 2, 3])
  for(y in [4, 5, 6])
    yield (x, y)
```

2.4.6 Local variables

In order to avoid polluting the Hobbes global namespace, we can declare variables as *local* to the current expression:

```
> let x = 9 in x * x
81
> x
stdin:1,1-1: Undefined variable: 'x' ...
```

In this case the name `x` is only in scope in the following expression. This allows us to re-use names without having to deal with `x1`, `x2`, etc.

Let expressions can allow more than one local variable to be declared:

```
> let x=1; y=10 in x + y
11
```

Indeed, let expressions are very powerful. In the following example we're first declaring, and then decomposing a tuple before the expression itself is evaluated:

```
> let f = (\x.(x,x)); (x, y) = f(20) in x + y
40
```

Notice how we're even able to reuse the name `x` across both the function declaration and the resultant tuple deconstruction. `Let` expressions are evaluated in declaration order, before the execution primary expression.

Let allows us to unpack tuple values in a convenient format:

```
> hostport = ("lndev1", 234)
> let (h, p) = hostport in show(p)
234
```

Note: This form of `let` is actually converted into a simple `match` for us by the Hobbes compiler:

```
match hostport with
| (h, p) -> show(p)
```

Note that this "de-sugaring" will only take place if the compiler can determine that the match can never fail.

Finally, let's wrap all that up with a match and a for comprehension:

```
> let start=1; end=4 in match [i | i <- [start..end], i % 2 == 0] with | [2, 4] ->
↪"evens" | _ -> "odds"
evens
```

2.5 Polymorphism

2.5.1 Type Classes

The hobbes approach to polymorphism is delivered through Type Classes, a way of externally declaring a piece of behaviour that a type can support. Type classes are a rich and powerful way of adding bits of functionality to existing types.

For example, all the numeric types support addition, and so I can declare a function using Hobbes's anonymous function syntax:

```
> add = (\u v x y z.u+v+x+y+z)
```

This can be read as "a function which takes arguments u, v, x, y, and z, and adds them all together". The backslash starts the function (or "lambda", because if you squint your eyes it looks a bit like the lowercase Greek letter "λ"), and the period separates the argument list from the function definition.

The types of the variables are left out, yet Hobbes will quite happily figure out types from the context in which they're used. In this case, we can say that the type of those values is "something which supports addition". Therefore, if we call 'add' with instances of numeric types, we'll get the answer we're looking for:

```
> add(0X01, 2, 3S, 4L, 5.0)
15
```

Note: Type inference in Hi

Hi has a number of advanced features, one of which is that it can show you the inferred type of an expression you've typed. We can use Hi to show us the inferred type of a smaller, simpler variant of our anonymous function:

```
:t (\y z.y+z)
Add a b c => (a * b) -> c
```

Hi has inferred the type for our three values (two parameters and one return value) to be the Type Class 'Add', and is showing the type of the function is one that takes a and b ("a * b") and returns c.

There are two parts, separated by the =>. It's easiest to take them backwards: The second part is the actual type of the value, which is `(a * b) -> c`. This can be read as "a function that takes an instance of `a`, and an instance of `b`, and returns an instance of `c`.

The *first* part is for type *restrictions*: things the compiler knows about `a`, `b` and `c` that limit what data instances of those types can represent. In this case, Hobbes is simply telling us that they must implement the `Add` type class (i.e. they overload the + operator).

Hobbes has simply inferred this about those types from the context in which they're used. This is in stark contrast to languates where types are restricted on what interfaces they implement.

Many other type classes are available in the *Hobbes standard library*. We've already seen an implementation of the equivalence typeclass `equiv`. Others include `multiply` (applied to types which have a `*`) and `print` (for types whose values can be printed).

2.5.2 Type annotations

`hi` is able to infer the type of a value:

```
> :t 3.2
double
```

We can also declare a polymorphic function and show that Hobbes can find restrictions on the type of its parameters. This is powerful for two reasons:

1. We don't have to write as much code;
2. We can share behaviour across data types as long as they share the ability to perform certain actions. That is, our algorithms become more generic.

```
> :t (\y.y+1)
Add a int b => (a) -> b
```

It's important to note that at runtime, all Hobbes functions are *monomorphic* - all the type parameters are resolved to the actual runtime type of the value and a specific function is output for evaluation.

Of course, in some situations we might want to declare a function or value with a specific type. In that case, we can use a *type annotation* to declare a value's type. This can give us extra type safety in cases where we don't want generic behaviour:

```
> i = 3 :: int
> j = 3.2 :: int
stdin:1,5-7: Cannot unify types: double != int
1 j = 3.2 :: int
```

We've specified that `j` is an int and the compiler has prevented us from assigning a double value to `j`.

Similarly, there's no silent upcasting:

```
> k = 3:: float
stdin:1,5-5: Cannot unify types: int != float
1 k = 3:: float
```

Type annotations allow us to specify types for functions which would otherwise be generic, using the type notations for functions that we saw before. In this case we don't need to specify type restrictions, we can just declare the value to be a function type from concrete type to concrete type:

```
> add1i = (\a.a+1)::int->int
> add1i(3)
4
> add1i(3.4)
stdin:1,1-6: Cannot unify types: int != double
1 add1i(3.4)
```

And because of the consistency of types:

```
> add1f = (\a.a+1)::int->float
stdin:1,9-28: Unsatisfiable predicate: Add int int float
1 add1f = (\a.a+1)::int->float
```

. . . because an `int` plus 1 is another `int`, *not* a `float`.

2.5.3 Type constraints

We can take this one step further:

```
> :t \x.x.Name
a.Name::b => (a) -> b
```

Remember that, in our lambda syntax, this can be read as “A function which takes x and returns x.Name” - i.e. the only thing we know about the type of x is that it has a member called Name. Hi will then give names to those two as-yet unnamed types: it calls them ‘a’ and ‘b’.

Note that Hobbes has inferred the type restriction on b: It’s whatever type the value of “a.Name” is. This function will work for *any* type that has a member called `Name`, which can be of any type!

2.6 The Embedded Compiler

Hobbes is a simple language with a rich type system, whose driving aim is to allow efficient reimplementation of functional logic at runtime.

As such, it comes packaged with a highly efficient compiler and type marshalling system.

2.6.1 Example: Hi, the Hobbes REPL

We’ve already seen one place where the Hobbes compiler is used. In its simplest form, the Hobbes REPL is a loop which takes Hobbes code, runs it line-by-line through the compiler, interprets the results and prints the output:

```
int main() {
  hobbes::cc c;

  while (std::cin) {
    std::cout << "> " << std::flush;
    std::string line;
    std::getline(std::cin, line);

    try {
      c.compileFn<void()>("print(" + line + ")")();
    } catch (std::exception& ex) {
      std::cout << "*** " << ex.what();
```

(continues on next page)

(continued from previous page)

```
    }

    std::cout << std::endl;
    hobbes::resetMemoryPool();
  }
}
```

2.6.2 Binding Functions

The Hobbes compiler allows us to *bind* a C++ function; to make it available by name in the Hobbes environment:

```
int addTwo(int i){
  return 2 + i;
}

...
c.bind("addTwo", &addTwo);
...
```

Note: Function Pointers

A key aspect of functional programming is that functions are just special kinds of *data*, and that whilst there may be operations which can be performed on data, the application which can be performed on a function is that of *parameter application*. Once a function has had all its parameters applied the code is executed and the return value given back to the callsite.

This has a two major implications: we can assign functions to values (as we've already seen), and we can write *higher-order functions*: those which take another function as a parameter, or return a function. In C++, the type of a function is denoted entirely by its signature, the abstract format for which is the function pointer:

```
int sum(int a, int b){
  return a + b;
}

int (*twoIntOperation)(int, int) = sum;
```

We declare `sum` as the concrete example of a function which takes two ints and returns an int, and assign it to the variable `twoIntOperation`. The type of `twoIntOperation` is exactly that: a function which takes two ints and returns one.

The syntax is a little hairy. In Hobbes, the function type syntax is a lot clearer, as we'll see very soon.

Once I've bound `addTwo` to the Hobbes compiler `c` it becomes available in the REPL environment. By invoking `addTwo`, I can see that Hobbes knows the type signature and knows how to invoke the function:

```
> addTwo
*** stdin:1,1-13: Failed to compile expression due to unresolved type constraint:
  Print (int) -> int
> addTwo(3)
5
>
```

Hobbes has a rich binding environment, with bi-directional support for marshalling common C++ types (as we've already seen, with int), as well as C++ structs, and the hobbes types such as record and variant.

2.6.3 Simple scalar types

As we've already seen, the Tuple is an important and common basic type in Functional Programming, used to keep elements of data together in a small lexical scope. Over time many of these functional types have 'leaked' into C++, where we now have `std::tuple` and the "two-tuple" special case, `std::pair`.

Hobbes is aware of the parametric `std::pair` and `std::tuple` types and composes them appropriately:

```
typedef std::pair<int, const hobbes::array<char>*> Writer;

Writer* getWriter(){
  return hobbes::make<Writer>(34, hobbes::makeString("Sam"));
}

...
c.bind("getWriter", &getWriter);
...
```

Then, in the Hobbes REPL:

```
> getWriter()
(34, "Sam")
```

2.6.4 Structs

There's usually a point at which our pairs or tuples grow in importance in our domain, and we want to give names to the members. In C++ we might use a struct for this purpose. Hobbes allows us to expose our C++ structs with the `DEFINE_STRUCT` macro:

```
DEFINE_STRUCT(Writer,
 (size_t, age),
 (const hobbes::array<char>*, name)
);

hobbes::array<Writer>* getWriters(){
  auto writers = hobbes::makeArray<Writer>(2);

  writers->data[0].age = 21;
  writers->data[0].name = hobbes::makeString("John");

  writers->data[1].age = 22;
  writers->data[1].name = hobbes::makeString("Paul");

  return writers;
}

...
c.bind("getWriters", $getWriters);
...
```

. . . and in the REPL:

```
> getWriters
age   name
---   ------
21    John
22    Paul
```

Note: Hobbes has been able to determine appropriate column names from the struct definition, just as it does with our hobbes-native record type!

2.6.5 Variant

Slightly more complex, our 'OR' type, the variant:

```
typedef hobbes::variant<int, const hobbes::array<char>*> CountOrMessage;

CountOrMessage* classify(int i){
  if(i<22){
    return hobbes::make<CountOrMessage>(i);
  }else{
    return hobbes::make<CountOrMessage(hobbes::makeString("haha"));
  }
}

...
c.bind("classify", $classify);
...
```

In our example we define our variant type in C++, and then create an instance depending on the value of some function parameter. Then in the Hobbes REPL we are able to call the bound function `classify` and deal with the result in a functional manner:

```
> classify(12)
|0=12|
> classify(42)
|1="haha!"|
```

2.6.6 Completing the round trip

In Hobbes we can expose higher-order functions simply. This allows us to expose Hobbes functionality in C++, completing the round trip!

As a Haskell-like language, the syntax is elegant. In the following example we declare a function `binaryIntFn` which takes two ints and returns an int. By compiling this function and binding it to the runtime Hobbes environment, we're able to "plug in" behaviour based on elements of the runtime environment:

```
int binaryIntFn(int (*pf)(int, int), int x){
  return pf(x, x);
}
```

```
>  binaryIntFn(\x y.x+y, 3)
6
> binaryIntFn(\x y.x*y, 4)
16
```

2.7 HLOG: The Logger

As well as a programming language, Hobbes also comes packaged with a whole array of functionality to allow highly efficient structured logging from within C++ applications.

2.7.1 Structured logging from within C++

Hobbes persistence is backed by a shared memory region which is initialised when the logging application starts up. In order to define some of the parameters for writing to this region, we'll use the `DEFINE_STORAGE_GROUP` macro.

DEFINE_STORAGE_GROUP

```
DEFINE_STORAGE_GROUP(
  MyGroupName,
  3000,
  hobbes::storage::Unreliable,
  hobbes::storage::AutoCommit
);
```

LogGroup correlation

Correlation of log messages is via the first parameter to the `DEFINE_STORAGE_GROUP` macro. The name must be unique - i.e. an application can't define two storage groups of the same name.

At the same time, we specify the number of shared memory pages to allocate for the storage group in the second macro parameter. This should reflect the maximum expected throughput for our log messages, because we need to take special action if we exceed the limit)

Reliability and drop-or-block

The third parameter to `DEFINE_STORAGE_GROUP` is the behaviour to exhibit if we reach the capacity of the ringbuffer storage backed by our group's shared memory region. `hobbes::storage::Unreliable` means that any attempts to write to a full buffer (as we'll see with `HSTORE` and `HLOG` below) will fail, while `hobbes::storage::Reliable` means that such writes will block until the buffer is serviced by a consumer.

Manual vs Automatic commit

The final parameter to `DEFINE_STORAGE_GROUP` allows us to specify the commit behaviour. `hobbes::storage::ManualCommit` means that all log statements up to the `commit` call will be grouped together in a transaction, which we can inspect later.

Alternatively, specifying `hobbes::storage::AutoCommit` gives us uncorrelated log messages on the consumer side, but we don't need to call `commit` ourselves.

Transactions which are manually committed will have a timestamp logged alongside them which can be used in out-of-band analysis and reporting, whereas autocommitted transactions have no timestamp. You can always call `commit` on an autocommit LogGroup, which will immediately persist the current transaction.

Finally, there's one more difference in how persisted data is made available to us in code - which we'll investigate in *logs and transactions*

Once the LogGroup has been set up, we can start to log data. For that, we'll want to look at the `HLOG` and `HSTORE` macros.

HLOG and HSTORE

The `HLOG` macro logs a formatted message to a given LogGroup with an event name. It can be used for logging string output from applications.

```
HLOG(
  MyGroupName,
  EventName,
  "Format String with positional arguments $0, $1",
  12
  42.0
);
```

Using the `HLOG` macro provides compile-time safety for the format string, and gcc will throw an error if you refer to a positional parameter that doesn't exist.

`HSTORE` allows for richer structured logging of data types that we declare. In the following example we're simply logging an `int`, but Hobbes has support for all the Hobbes primitive types, as well as `std::string`, `char *`, and arrays, vectors and tuples of its supported types.

```
HSTORE(
  MyGroupName,
  EventName,
  12
);
```

Information about the types Hobbes is able to persist can be found in *hobbes persistable types*

Warning: File size

Practically speaking, what we're discussing here is data *persistence* rather than logging. For that reason there's no model for output file rotation. That means that your persisted data files might grow very large in size, and you'll need to find a way to externally manage that.

At Morgan Stanley, Hobbes persistence is used in production applications which might be bounced daily or even weekly, resulting in persisted files of many dozens of gigabytes.

Declaring structured types for HSTORE logging

We can create our own structured types for HSTORE logging with the `DEFINE_HSTORE_STRUCT` macro:

```
DEFINE_HSTORE_STRUCT(
  StructName,
  (double, val1),
  (int, val2)
);
```

2.7.2 Example

A simple example of a log producer is shown below. We initialise a small logger with drop and auto-commit semantics, and then drop a few log messages with HLOG:

```
#include <hobbes/storage.H>
#include <chrono>
#include <thread>

using namespace std;
using namespace std::chrono;
using namespace hobbes::storage;

DEFINE_STORAGE_GROUP(
  SimpleLogger,
  1,
  Unreliable,
  AutoCommit
);

int main() {

  while(true){
    HSTORE(SimpleLogger, FirstEvent, "First", 0, 1, 2);
    HSTORE(SimpleLogger, SecondEvent, "Second", "data", 3.4);

    this_thread::sleep_for(milliseconds(500));
  }
}
```

A worked example of a log producer in C++ can be found in the *examples*

2.8 Data Types

What types of data can be logged in Hobbes?

2.8.1 Primitive types

Essentially, whatever you'd expect. Because the persistence format is binary, it's both space-efficient and type safe, meaning the datatype you persist is what you get out. As such, the following primitive types can be persisted:

```
bool
uint8_t
char
16, 32, and 64 bit signed and unsigned ints
single and double precision floats
std::string and const char*
```

2.8.2 Aggregate types

If a type `T` can be persisted, then its array type `T[]` and its vector `std::vector<T>` can be persisted.

If types `T0, T1, ..., Tn` can be persisted, then a tuple `std::tuple<T0, T1, ..., Tn>` can also be persisted.

2.8.3 Custom types

You can create a struct that Hobbes can log using the `DEFINE_PACKED_HSTORE_STRUCT` macro, as long as the member types are persistable:

```
DEFINE_PACKED_HSTORE_STRUCT(TemperatureAndPressure,
  (double, Temperature),
  (double, Pressure)
);
```

With a little work, you can even make a custom class loggable as a field by specialising the `hobbes::storage::store<T>` type and implementing the static functions found inside.

You'll find `store` in `include/Storage.h` in the github repo.

2.9 Consuming logs with hog

When our producer code runs and starts logging, it initialises a shared memory region of the appropriate size and with a name which is visible to Hobbes logging consumers.

Over time, as the application logs, the ringbuffer which backs onto the shared memory region will fill up, and depending on the reliability semantics specified in the `DEFINE_STORAGE_GROUP` invocation, further writes will either block or fail.

To prevent this from happening, a performant consumer must service the queue and provide further processing for log messages. One such consumer which is pre-written with some solid default behaviour is `hog`.

2.9.1 hog

```
$ hog
hog : record structured data locally or to a remote process

  usage: hog [-d <dir>] [-g group+] [-p t s host:port+] [-s port] [-c] [-m <dir>]
where
  -d <dir>           : decides where structured data (or temporary data) is stored
  -g group+          : decides which data to record from memory on this machine
  -p t s host:port+ : decides to send data to remote process(es) every t time units␣
↪or every s uncompressed bytes written
  -s port            : decides to receive data on the given port
  -c                 : decides to store equally-typed data across processes in a␣
↪single file
  -m <dir>           : decides where to place the domain socket for producer␣
↪registration
```

Hog is reponsible for consuming messages from a particular LogGroup and coordinating their onward flow. Options are:

1. Write messages to local disk. The default logfile directory is `./$groupname`, and this can be overridden with `-d`.
2. Write messages to a remote hog process, running on a server described with `-p`

Conversely, Hog is also able to receive log messages from a remote process with `-s`.

2.9.2 Logging to disk with hog

If we take our log message driver application from the *previous section*, start a Hog listener on the same LogGroup and then start the driver, we'll see something like the following:

```
$ hog -g SimpleLogger
[2018-01-01T09:00:00.867323]: hog running in mode : |local={ dir="./$GROUP/$DATE/data
↪", serverDir="/var/tmp" groups={"SimpleLogger"} }|
[2018-01-01T09:00:00.867536]: install a monitor for the SimpleLogger group
[2018-01-01T09:00:01.637614]: new connection for 'SimpleLogger'
[2018-01-01T09:00:01.733374]: queue registered for group 'SimpleLogger' from␣
↪3817:3817, cmd 0
[2018-01-01T09:00:01.912325]:  ==> FirstEvent :: () (#0)
[2018-01-01T09:00:01.969274]:  ==> SecondEvent :: [:char|10L:] (#1)
[2018-01-01T09:00:02.009241]:  ==> log :: <any of the above>
[2018-01-01T09:00:02.129401]: finished preparing statements, writing data to './
↪SimpleLogger/2018.01.01/data.log'
```

This output tells us a couple of things:

1. Firstly, Hog is running in local mode, meaning that it's going to consume messages from the Hobbes ringbuffer in memory and write them out to disk;
2. When the driver application starts we get some state information about the LogGroup, and the message types;
3. Hog has been able to determine the message names and, crucially, their types.

2.9.3 Reading logs from remote processes with hog

You can use two instances of hog to send Hobbes log messages from one host to another. The setup is simple:

On the logging host

On the host where the application is logging, invoke `hog` with the `-p` parameter as follows:

```
$ hog -g SimpleLogger -p $t $s $h:$p
```

Where:

- `$t` is the time in seconds between sent messages
- `$s` is the buffer size in *uncompressed* bytes before which a message is sent
- `$h:$p` is the host and port on which the receiving instance of hog is running.

Note: Messages are sent to the receiving host *at least* as often as `$t`. If `$s` bytes are ready to be sent, this will happen regardless of the time since the last message.

For example:

```
myhost $ hog -g SimpleLogger -p 1 100 anotherhost:10101
[2018-09-26T15:03:12.126732]: hog running in mode : |batchsend={ dir="./$GROUP/$DATE/
↪data", serverDir="/var/folders/pp/g8vs2j61015fsy8lqlbh_8tm0000gn/T/", clevel=6,␣
↪batchsendsize=100B, sendto=["anotherhost:10101"], groups={"SimpleLogger"} }|
[2018-09-26T15:03:12.128300]: install a monitor for the 'SimpleLogger' group
```

On the receiving host

On the receiving host, the setup is very similar to the first case, “Logging to disk”. The only difference is that rather than reading messages from the in-memory ring buffer, messages are instead received on a local port.

The setup is simple:

```
$ hog -g SimpleLogger -s $p
```

Where:

- `$p` is the port on which messages are to be received.

For example:

```
anotherhost $ hog -g SimpleLogger -s 10101
[2018-09-26T15:03:52.277627]: hog running in mode : |batchrecv={ dir="./$GROUP/$DATE/
↪data", localport=10101 }|
```

Execute the logging driver

Finally, on the logging host, execute your log driver application. You’ll see both of the hog processes spring to life, and the receiving host will print the type information of the two messages we sent.

2.10 Processing logs with hi

We’ve already seen how to write C++ processes which log with the Hobbes logging API, and how we can use hog to pick up those log messages from shared memory and either store them locally or forward them on to another hog process running on a remote host.

In this chapter, we’ll look at how we can read those persisted logs and process them in the Hobbes environment.

You can read a logfile as it’s being written to, or one that’s been written to in the past. The name of the logfile is mentioned in the first few lines of the output from Hog. You’ll see something like

```
finished preparing statements, writing data to './SimpleLogger/2018.09.26.data-0.log'
```

For simplicity, we’ll use hi, the Hobbes REPL.

2.10.1 Reading the file

```
$ hi
> messages = inputFile :: (LoadFile "./SimpleLogger/2018.09.26/data-0.log" w) => w
```

That’s how we open the file in hi. It looks a bit cryptic, so let’s try to unpack it a little bit.

Firstly, this is simply an assignment of something called `inputFile` to the name `messages`. The interesting bit is in the type annotation, which you might remember from *this previous section*. This one is a little more long-winded than what we’ve seen before - but that’s essentially what it is: a tighter specification for the *type* of what `messages` is.

And that’s interesting, because it’s a gentle hint at the fact that `messages` is a *type-safe* collection of all the structured data inside the file we’ve specified there, `data.log`!

We’ll come back to how that works a little further down, but right now we need to introduce the Hobbes logfile format.

Logfile format

Hobbes stores its persisted data in a binary format, which means the files it outputs aren't human readable. This makes them very space-efficient and means it's very quick for Hobbes to do file I/O, it just means you have to use tooling to read the files.

In practice, that's not so much of a problem because you either write the tools in advance, or else read a logfile in `hi` and use the appropriate queries to find the data you're looking for. For example, at Morgan Stanley there's a rich library of Hobbes tools which are used to hunt through logfiles to answer common queries.

The logfiles themselves have a *header/body* format, with a header describing the names and types (and by inference, the *sizes*) of each column. Immediately following the header is the body, which contains the actual persisted data. Therefore, performing an equality search on a member *n* of a struct of size *m* is trivial: you simply loop through every *m* bytes and compare the search value against the value at an offset of *n*.

The LoadFile unqualifier

This brings us back to `LoadFile`, and the type annotation in the code above. By now, you might have figured out what's actually happening. The syntax is inherited from Haskell, but what `LoadFile` is doing is this:

1. Loading the file listed as its argument
2. Reading the header and extracting the type information for each column
3. Externalising the type information into the annotation

Therefore the annotation becomes, at compile time, a *type-safe* indication of the type of data contained in the file. In a sense, LoadFile is a hook into the Hobbes compiler, allowing the hi process to create a type at compile time, based on the type information available in the file header.

Note: ORM: an analogy

If all this is a bit too hairy - don't worry! You can always just type the code and it'll work. Simply replace the name of the file with that of your persistence file and you're good to go.

If you'd like a gentle push, you can think of `LoadFile` as a kind of backwards ORM. An Object-Relational Mapping tool can create a database schema for you based on a set of classes defined in code. You write your code and construct objects, and based on the ORM rules, these objects can be trivially persisted in a database table according to the schema.

Well, what if that worked backwards? A reverse ORM tool could look at a database table and read the schema, and construct classes that you can code against. That's exactly what *LoadFile* is doing - except it all happens automatically, as part of the compilation process.

Incidentally, the *same* process is happening when we use *hog* to send structured data over the network for persistence: the header is sent when the target hog process starts up, and the type information contained into the header is passed through to the persisted file before data starts to stream.

2.10.2 One-off data processing

We can query data inside a logfile using the comprehensions syntax we saw *previously*. It's even possible to see updates to the file as it's being written - changes will be visible in hobbes code whilst elements are being added to the file. This allows for reactive programming but it also means that calculation results may change between invocations - so be careful!

As a collection

Firstly, we can see what data is available in the file simply by typing the name of the variable, *messages*:

```
> messages
```

This shows us all the data members available to us, along with their types. You might recognise this type information from the *hog* output, when we saved the persisted data in *previous section*:.

Note: types There's a lot going on here! For a start, you might notice the types of FirstEvent and SecondEvent don't look all that much like you'd expect. For example, remember that our logger contains the following line:

```
HSTORE(SimpleLogger, FirstEvent, "First", 0, 1, 2);
```

You might reasonably expect *messages.FirstEvent* to have the type `[([char]*int*int*int)]` - i.e. an array of tuples, each containing a string and three ints. The internal representation of the Hobbes persistence file is just slightly out of scope for this introduction - but you'll be pleased to know it (mostly) doesn't matter that much: as we'll see soon, you can use the Hobbes comprehension syntax to deal with the data as though it looked just as you expect!

Both datasets (one for each event) are available to us as data members under the file variable `messages`:

```
> messages.FirstEvent
First 0 1 2
First 0 1 2
First 0 1 2
First 0 1 2
First 0 1 2
First 0 1 2
First 0 1 2
...
```

This is useful to show that persistence is working. However, in order to process the data, you'll probably want to make use of Hobbes's comprehensions.

As a comprehension

Although the type of the data isn't quite an array, we can use comprehension syntax to collect, organise, and process the persisted data. In fact, this is a very common usecase for Hobbes in production. It allows us to filter and map across large amounts of data in a neat consistent manner:

```
> [ x.1 | x <- messages.FirstEvent]
[0, 0, 0, 0, 0, 0, 0,
...
```

Note: tuples?

Hobbes exposes this persisted element (the *line* of logged data, really) as a tuple, so you can unpack it using the numbered indexing syntax. In this case, we're showing the (zero-indexed!) first field - i.e. the 0 from the log message above.

We can take this further and unpack the tuple in the extraction portion of the comprehension:

```
> [ (x, y) | (x, y, z, a) <- messages.FirstEvent ]
"First" 0
"First" 0
"First" 0
"First" 0
"First" 0
"First" 0
"First" 0
```

Here we’re unpacking all four fields from the log statement and printing the first two.

Take a slice

Similarly, we can use the “slice” notation to work with a subset of logged messages:

```
> messages.SecondEvent[0:3]
First 0 1 2
First 0 1 2
First 0 1 2
```

Note: ordering

Due to the internal structue of the persisted file, while elements may *look* ordered, they are not. We can force a ‘most recent first’ ordering of logged elements using the open-slice notation:

```
messages.SecondEvent[0:]
```

Note: where’s my data?!

If you have a process which is logging and you’re not seeing any updates, it might be that you’re writes from the sending process are being batched and haven’t yet been flushed.

This can be the case if you’re not logging much data, and using auto-commit persistence in your *storage group definition*.

If that’s the case, you can force a flush by calling the group’s `.commit()` member in your logging code.

e.g. for a storage group called *SimpleLogger* (like ours has been), you’d call

```
SimpleLogger.commit();
```

2.10.3 Logs and Transactions

As we discussed above, you can find the names and types of the log events available in the persisted storage group file simply by inspecting the variable you assigned it to using *LoadFile*.

```
> messages
```

Alongside the data members for each of your log events, you’ll also see a `statements` field and a `log` field. `statements` is for Hobbes internal use (feel free to have a look - it contains some back-references to the logging source code), whilst `log` is a collection of all messages logged to any event connected to the group.

This can be useful if you want a stream of all the messages you've logged, and the data is available in a handy type that Hobbes has created for us - a variant of all the possible logged types. This makes it easy to use the *pattern matching* syntax to iterage through logged messages and act on them.

Manually committed transactions

As we touched on in *logging*, log data persisted in a *manually committed* log group will have a timestamp associated with each entry.

This is simply an artefact of the way Hobbes has been used at Morgan Stanley, rather than an explicit design decision. If you wanted a timestamp associated with an auto-commit group you could specialise `hobbes::storage::store<T>` for `std::chrono::time_point` (see *data types* for more information) and collect a timestamp member for each log statement.

However, there's one more difference - for a manually committed group there's a `transactions` member instead of a `log` member for your group's persistence file, which shows the timestamp alongside the log data.

2.10.4 Reactive processing

Once we have a reference to a Hobbes file, we can perform realtime analysis of the data it contains with the `signals` API. If new data is written to the file, this event handler will be called:

```
> signals(messages).FirstEvent <- (\_.do { putStrLn("message received!"); return true␣
↪})
> message received!
message received!
message received!
[...]
```

This allows us to do reactive programming across Hobbes processes.

2.11 Networking with hi

Hobbes supports native, typesafe client-server network programming in a very similar manner to its support for structured logfiles. Connection information alongside the types of data involved are exposed through an unqualifier much like *LoadFile*, in a manner mostly invisible to the user.

You can use Hobbes networking to perform actions on another host, such as gathering usage statistics, or performing administrative actions such as changing the log level.

2.11.1 Setting up the receiver

hi can be set to receive messages over the network. If we invoke *hi* with the `-p` flag we can specify the port to listen on:

```
$ hi -p 8080
[...]
running a repl server at myhost:8080
>
```

We'll call this the *server*, and next we'll connect to it over the network.

2.11.2 Opening a connection

From another instance of *hi*, create a connection to the *server* using the `Connect` unqualifier:

```
$ hi -s
> c = connection :: (Connect "myhost:8080" p) => p
```

Note: Unqualifiers

For more information about how this works, have a look at the *LoadFile unqualifier*, which we use to load data using the Hobbes persistence API.

We'll call this the *client*. From here, you can inspect the details of the connection with `printConnection`:

```
> printConnection(c)
id expr input output
-- ---- ----- ------

>
```

There's not much here yet, so let's create some functionality on the *server*.

2.11.3 Remote methods

Back on the *server*, create a new function called addOne:

```
> addOne = \x.x+1
>
```

Then on the *client*, let's make that functionality available remotely. We'll use the hobbes type negotiation mechanism to make sure all the types line up:

```
> receive(invoke(c, `addOne`, 12))
13
```

Wow! There's some new Hobbes here, so let's go through this line piece by piece.

The `invoke` and `receive` functions allow Hobbes to execute commands remotely and interpret the results. We call them together becuase the type information about the return value of the `addOne` function is passed between them.

Secondly, the strange 'quoted' form of `addOne`. The quoted form of the invocation isn't a string - it's parsed but as-yet unexecuted Hobbes in a form which can be serialised and set to a remote Hobbes process for invocation. It's this mechanism that Hobbes uses to determine the return type of the method - on the remote process!

In this manner we're able to execute functions on the server from the client - without any of the complex type negotionation or serialisation that we'd otherwise have to do.

Once the work has been executed remotely, the result has been serialised, sent across the network, deserialised and displayed in our local *client* prompt.

2.11.4 Inspection of the connection

If we use `printConnection` to take another look at `c`, we'll see that the initial remote invocation of the function `addOne` has had some effects:

```
> printConnection(c)
id expr input output
-- ---- ----- ------
1  int  int   addOne
```

Firstly, we can see that Hobbes has given the remote `addOne` function a numeric ID - this means that future invocations will be much faster.

Secondly, Hobbes has used the connection to communicate with the remote host and find out the type of `addOne` - a function that takes an `int` and returns an `int`.

2.11.5 Delayed Invocation

In the above example the type information Hobbes gathered from the server was made available at the first invocation of the method, using `receive`. However, Hobbes has the ability to query type information from the server using the unqualifier mechanism much earlier, before the method is even invoked.

Go back to the *server* and add another method, addTwo:

```
addTwo = \x.x+2
```

Then on the *client*,

```
> remoteAddTwo = \x.receive(invoke(c, `addTwo`, x::int))
> printConnection(c)
id expr input output
-- ---- ----- ------
1  int  int   addOne
2  int  int   addTwo
```

In this example, `remoteAddTwo` is a function defined by a lambda - that we haven't yet called! All we've passed is the information about the input type - the `int` argument to `addTwo` - and the Hobbes server process has done all the type inference and returned the structured type data for us.

We can invoke the remote function in the usual way, by passing parameters to the function name:

```
> remoteAddTwo(3)
5
```

2.11.6 Errors

Because all the type information is evaluated on the remote host, any processing errors or type mismatches will also come from the other server. For example, try to invoke a function that doesn't yet exist:

```
> receive(invoke(c, `addSeven`, 3))
stdin:1,1-33: Error from server: stdin:0,0-0: Undefined variable: 'addSeven'
```

2.12 Data Types

The rules for which built-in types can be used in Hobbes networking are the same as for *Logging datatypes*.

For custom datatypes, you should specialize `hobbes::net::io<T>`, or refer to `hobbes/net.h` in the GitHub repo.

2.13 C++ Bindings

In addition to the native support Hobbes has for networking, there's a useful C++ API that you can use to send messages to Hobbes processes.

2.13.1 DEFINE_NET_CLIENT

```
#include <iostream>
#include <hobbes/net.H>

DEFINE_NET_CLIENT(
  Connection,                             // Profile name, which we'll use later
  (mul, int(int,int), "\\x y.x*y")   // Functionality to evaluate remotely
);
```

Similarly to the Logging code, we first define the Connection semantics that we want to use. This macro creates a connection class for us which has a constructor and a data member for the `mul` function, along with all the type negotiotiation and connectiona management logic.

2.13.2 Using the connection

```
int main(int, char**) {
  try {
    Connection c("localhost:8080");

    std::cout << "c.mul(1,2) = " << c.mul(1,2) << std::endl;

  } catch (std::exception& e) {
    std::cout << "*** " << e.what() << std::endl;
  }
  return 0;
}
```

Here we're instantiating the synthesised `Connection` class, invoking its `mul` member (remotely!) and printing out the result.

2.13.3 Running the code

Again, spin up an instance of *hi* listening on port 8080. You won't see any output from here.

```
$ hi -s -p 8080
```

Next, simply run your c++ driver program!

```
$ ./test
c.mul(1,2) = 2
```

2.14 The Standard Library

As well as the compiler infrastructure, Hobbes comes packed with a rich Standard Library of functions and types that make it easy to get started.

They're all available in the default namespace, and we've used quite a few of them already throughout the documentation.

In this section, we'll draw your attention to some of the most common and useful elements of the Hobbes Standard Library.

All the members here are written in plain, vanilla Hobbes - so digging into the source on GitHub is a great way to learn the language!

2.14.1 Simple Arithmetic

These Type Classes, and their instances (what we'd call a 'realisation' of the type class for a concrete type) allow Hobbes to handle arithmetic in a polymorphic manner:

```
class Add a b c | a b -> c where
  (+) :: (a,b) -> c

instance Add int int int where
  (+) = iadd

instance Add long long long where
  (+) = ladd

...
```

Because the `Add` Type Class (and the rest of the family: `Subtract`, `Multiply`, and `Divide`) is available in the global namespace, I can use its instances implicitly - i.e., the + operator is defined for all the basic types:

```
> 1 + 2
3
```

Note: built in?

It's important to note that the + operator isn't "built in" to Hobbes as operators might be in other programming languages. The resolution of + works because the compiler has recognised that there is a typeclass `Add` which provides an operator called + that can act on two ints.

Indeed, we can support + elsewhere by instantiating the Type Class for our own types:

```
type counter = { count: int}

counterAdd = (\x y. { count = iadd(x.count,y.count)})

instance Add counter counter counter where
  (+) = counterAdd
```

Note: type definitions in hi

Just like in Haskell, the hi REPL doesn't support type definitions. So in this case, we've included the above Hobbes code in a file called `counter.hob` and instructed the *hi* REPL to read it at startup. The members defined in `counter.hob` can then be used directly.

```
$ ./hi counter.hob
loaded module 's.hob'

> {count = 22} + {count = 33}
55
>
```

2.14.2 Maybe

`Maybe` is deeply idiomatic in functional programming, and is starting to seep into more popular languages too.

Effectively, it's a clean handling of the case where a function may legitimately not be able to provide a value.

Note: Maybe

Consider a function `getUser([char]) -> User`, which gets user information from some external source, when given the users name.

What would you expect the function to return if a user with that name was not found? It's an entirely reasonable situation and not really *exceptional* at all.

Indeed, in some languages you'd expect the function to throw a UserNotFoundException, which you'd have to catch and deal with outside the regular flow of control.

In Functional Languages, we'd change the signature of the function to return a `Maybe` type.

The `maybe` is really a sum type that looks like this:

```
(()+a)
```

That is, it can either have a value, or it can have no value. You instantiate `maybe` with the two constructors `just` and `nothing`:

```
> maybenums = [just(1), just(2), nothing, just(3)]
maybenums = [1, 2, , 3]
> dropNulls(maybenums)
[1, 2, 3]
```

Maybe has a large number of utility functions that deal with either case - i.e. where there is a value and where there is no value.

```
> map((\x.x+2), maybenums)
[3, 4, , 5]

fromMaybe(0, maybenums[1])
> 2

fromMaybe(0, maybenums[2])
> 0
```

In the above example we're extracting the value from the `maybe` if one exists, or else we're providing a sensible default (in this case, the integer `0`).

2.15 hi, the Hobbes interpreter

In the `bin` directory of your Hobbes build, you'll find the Hobbes interactive interpreter, "hi". You can use it to execute much of the code you see on these pages by just typing the line and hitting 'enter'. It's an example of a REPL, or "Read, Eval, Print Loop" - because those are the key steps that allow it to execute the code you enter.

When used interactively, Hi will show you the results of the execution of a line immediately:

```
hobbes/bin $ hi

hi : an interactive shell for hobbes
  type ':h' for help on commands

> true
true
> 1 + 2.3
3.3
```

Throughout this documentation we'll show the Hi prompt ('>') to indicate text you can type into Hi. You don't need to type the prompt too.

2.15.1 hi can read files

If you pass the name of a text file to the Hobbes interpreter hi, it'll evaluate the code in the file and make symbol names available to you in the hi session:

```
bin/hobbes $ cat match.hob
foo = match 3 with
  | 1 -> "hello"
  | 2 -> "hobbes"
  | _ -> "oops"

bin/hobbes $ hi match.hob
hi : an interactive shell for hobbes
  type ':h' for help on commands

loaded module 'match.hob'
> foo
"oops"
```

This makes it much easier to write more complex, multi-line expressions of the kind you'll see in later sections. Some of these examples are therefore shown without the hi prompt.

2.15.2 Type information

Hobbes is a strong, statically-typed language, meaning every expression has a type which is known at compile time. The Hobbes compiler makes this type information available to the REPL Hi.

`:t` shows us the Hobbes type of the defined function:

```
> add = (\a b.a+b)
> :t add
Add a b c => (a * b) -> c
```

In this case, Hi shows us that we've implicitly defined `add` as an implementation of the typeclass `Add`.

Digging a little further, we can use `:c` to see the implementation of the `Add` typeclass, which is defined as any type which has an + member:

```
> :c Add
class Add | #0 #1 -> #2 where
  + :: (#0 * #1) -> #2
```

We can see all implemetations of a typeclass with `:i`:

```
> :i Add

instance Add char char char where
  + = cadd::(char * char) -> char::(char * char) -> char
instance Add byte byte byte where
  + = badd::(byte * byte) -> byte::(byte * byte) -> byte
instance Add short short short where
  + = sadd::(short * short) -> short::(short * short) -> short
...
```

2.15.3 Timing information

Some timing data can be retrieved for the complation and evaluation phases of a Hobbes expression:

```
> :e add(1, 3)
average over 1000 runs: 43ns
minimum runtime: 42ns
maximum runtime: 131ns
```

2.15.4 Commandline options

-s	Silent mode - don't display the prelude or version information
-p nnnn	Listen on port *nnnn*

2.16 A Simple REPL

In the Embedding Hobbes chapter we built a simple Hobbes interpreter with two-way function bindings.

This allowed us to call C++ functions from within the Hobbes environment, and also, via the function pointer, to execute Hobbes code in the containing C++ code.

Here's the full code listing. Afterwards we'll dig into all the parts one by one:

```
#include <iostream>
#include <stdexcept>
#include <hobbes/hobbes.H>

int addTwo(int i){ return 2 + i; }

DEFINE_STRUCT(Writer,
  (size_t, age),
  (const hobbes::array<char>*, name)
```

(continues on next page)

(continued from previous page)

```
);

typedef std::pair<int, const hobbes::array<char>*> Writer;

Writer* getWriter(){
  return hobbes::make<Writer1>(34, hobbes::makeString("Sam"));
}

hobbes::array<Writer>* getWriters(){
  auto writers = hobbes::makeArray<Writer>(4);

  writers->data[0].age = 21;
  writers->data[0].name = hobbes::makeString("John");

  writers->data[1].age = 22;
  writers->data[1].name = hobbes::makeString("Paul");

  writers->data[2].age = 22;
  writers->data[2].name = hobbes::makeString("George");

  writers->data[3].age = 42;
  writers->data[3].name = hobbes::makeString("Sam");

  return writers;
}

typedef hobbes::variant<int, const hobbes::array<char>*> CountOrMessage;

CountOrMessage* classify(int i){
  if(i < 22){
    return hobbes::make<CountOrMessage>(i);
  }else{
    return hobbes::make<CountOrMessage>(hobbes::makeString("haha!"));
  }
}

int binaryIntFn(int (*pf)(int, int), int x){
  return pf(x, x);
}

int main() {
  hobbes::cc c;

  c.bind("addTwo", &addTwo);
  c.bind("getWriters", &getWriters);
  c.bind("getWriter", &getWriter);
  c.bind("classify", &classify);
  c.bind("binaryIntFn", &binaryIntFn);

  while (std::cin) {
    std::cout << "> " << std::flush;

    std::string line;
    std::getline(std::cin, line);
    if (line == ":q") break;

    try {
```

(continues on next page)

(continued from previous page)

```
      c.compileFn<void()>("print(" + line + ")")();
    } catch (std::exception& ex) {
      std::cout << "*** " << ex.what();
    }

    std::cout << std::endl;
    hobbes::resetMemoryPool();
  }

  return 0;
}
```

2.16.1 Sections

Prelude

```
#include <iostream>
#include <stdexcept>
#include <hobbes/hobbes.H>
```

First comes the include statements for the c++ pre-processor. The Hobbes header file is available in the GitHub repo under the *include* directory.

A simple function

```
int addTwo(int i){ return 2 + i; }
```

This simple c++ funciton takes an int and adds 2 to it, returning the result. It's used later, as an example of binding a c++ funciton in to the hobbes environment.

Binding custom datatypes

```
DEFINE_STRUCT(Writer,
  (size_t, age),
  (const hobbes::array<char>*, name)
);

typedef std::pair<int, const hobbes::array<char>*> Writer1;

Writer* getWriter(){
  return hobbes::make<Writer>(34, hobbes::makeString("Sam"));
}
```

Here, we use the `DEFINE_STRUCT` macro to make Hobbes aware of a custom datatype, and use the `make*` functions for our custom type and for `std::string`. This makes Hobbes responsible for allocating space for this data, and allows it to deallocate the memory when it's finished.

Arrays

```
hobbes::array<Writer>* getWriters(){
  auto writers = hobbes::makeArray<Writer>(4);

  writers->data[0].age = 21;
  writers->data[0].name = hobbes::makeString("John");

  writers->data[1].age = 22;
  writers->data[1].name = hobbes::makeString("Paul");

  writers->data[2].age = 22;
  writers->data[2].name = hobbes::makeString("George");

  writers->data[3].age = 42;
  writers->data[3].name = hobbes::makeString("Sam");

  return writers;
}
```

We're extending the example to include `makeArray`, showing how Hobbes can interact with collections.

Variants

```
typedef hobbes::variant<int, const hobbes::array<char>*> CountOrMessage;
```

Firstly, we use a c++ *typedef** to give a nice name to a Hobbes variant

```
CountOrMessage* classify(int i){
  if(i < 22){
    return hobbes::make<CountOrMessage>(i);
  }else{
    return hobbes::make<CountOrMessage>(hobbes::makeString("haha!"));
  }
}
```

Hobbes creates factory methods for each subclass in the variant, and so, depending on some external factor, we're able to initialise either a *count* or a *message*.

Function pointers

```
int binaryIntFn(int (*pf)(int, int), int x){
  return pf(x, x);
}
```

The basic mechanism by which work is abstracted, and how we can externalise behaviour from Hobbes - allowing us to interact with Hobbes functionality from outside the environment. In this case we expect `binaryIntFn` to be called with two items - firstly, a function which takes two `int``s and returns an ``int`, and secondly an `int`.

The result of the application of the function with the second argument twice is then returned to Hobbes as an `int`.

2.16.2 main()

Finally, the `main` method brings it all together:

```
int main() {
  hobbes::cc c;

  c.bind("addTwo", &addTwo);
  c.bind("getWriters", &getWriters);
  c.bind("getWriter", &getWriter);
  c.bind("classify", &classify);
  c.bind("binaryIntFn", &binaryIntFn);

  while (std::cin) {
    std::cout << "> " << std::flush;

    std::string line;
    std::getline(std::cin, line);
    if (line == ":q") break;

    try {
      c.compileFn<void()>("print(" + line + ")")();
    } catch (std::exception& ex) {
      std::cout << "*** " << ex.what();
    }

    std::cout << std::endl;
    hobbes::resetMemoryPool();
  }

  return 0;
}
```

1. We initialise an instance of `hobbes::cc` on the stack;
2. We bind five functions to it by their address;
3. Then, in a loop, we print a prompt, accept a string from STDIN, and attempt to compile and execute the input string;
4. Any failures are caught and reported;
5. Finally, any unreferenced data left after the REPL loop is collected to avoid memory leaks.

2.17 A Simple Logger

```
#include <hobbes/storage.H>
#include <chrono>
#include <thread>

DEFINE_STORAGE_GROUP(
  RaindropLogger,
  1,
  Unreliable,
  ManualCommit
  );

```

(continues on next page)

(continued from previous page)

```
int main(){
  while(true){
    auto raindrop_size { rand () % 10 };

    switch( rand() % 5){
      case 0: HSTORE(RaindropLogger, FirstBucket, size);
              break;
      case 1: HSTORE(RaindropLogger, SecondBucket, size);
              break;
      case 2: HSTORE(RaindropLogger, ThirdBucket, size);
              break;
      case 4: HSTORE(RaindropLogger, FourthBucket, size);
              break;
    }

    RaindropLogger.commit();

    cout << "." << flush;
    this_thread::sleep_for(milliseconds(500));
  }
}
```

In this example we're collecting raindrops of different sizes in four different buckets, and every time a raindrop falls in a bucket, we're logging that information.

We therefore have a storage group called "RaindropLogger" with unreliable manual-commit semantics (i.e. we'd rather lose an update than block on the publishing side, and writes to the persistence file are going to happen when we call `flush`.

There are four event *names*, one for each bucket, and a drop falls every half a second. There's a one in five chance that a drop won't fall in any of the buckets, just to make things interesting!

2.17.1 Running locally

Fire up an instance of *hog* and let it listen locally for updates to the RaindropLogger group:

```
$ hog -g RaindropLogger
[2018-11-07T14:19:20.812276]: hog running in mode : |local={ dir="./$GROUP/$DATE/data
↪", serverDir="/var/tmp", groups={"RaindropLogger"} }|
[2018-11-07T14:19:20.812276]: install a monitor for the 'RaindropLogger' group
```

In another shell on the same machine, build and run your RaindropLogger driver:

```
$ ./RaindropLogger
..............
```

You'll see the *hog* output change to include the name of the written logfile. Follow the instructions in *Processing logs with hi* to read the file into the *hi* REPL

2.17.2 Queries

Let's jump into *hi* again and play with our data. We can use the `size` function to see how many raindrops fell in a given bucket:

```
$ hi
> raindrops = inputFile :: (LoadFile "log_file_name" w) => w
> size(raindrops.FirstBucket)
51
```

Get the total amount of water that fell in the third bucket:

```
> sum(raindrops.ThirdBucket[:0])
614
```

Note: Slice?

Why do we have to use the slice notation in the above example?

Great question! If you use `:t` to inspect the type of *FirstBucket* you'll see that whatever it is, it's *not* an array!

For performance reasons in order to allow the size of the group to grow over time whilst it's being monitored in memory, the type of a log group is a *stack of arrays*, meaning special functions need to be used to work with persisted messages.

That's why the slice notation works but array indexing doesn't. It also explains why we're using the slice notation here: it's to force the log data into an array:

```
> :t raindrops.ThirdBucket[1:2]
[int]
```

The `sum` function is overloaded for arrays, but not for the Hobbes internal *stack of arrays* type!

Let's get the count of all raindrops bigger than size 5:

> size([x | x <- raindrops.FirstBucket[0:] ++ raindrops.SecondBucket[0:] ++ raindrops.ThirdBucket[0:] ++ raindrops.FourthBucket[0:], x > 5]) 193

Transactions

Because we used a *manually* commited log group for our raindrop data (see *Logs and Transactions*), we can see a list of all the raindrops in order, regardless of the bucket they fell in:

```
> raindrops.transactions
```

This even shows us the committed transactions for where there was no logged event - i.e. the one in five chance that a raindrop missed a bucket!

2.18 About

2.18.1 Contributing

Contributing to Hobbes is easy.

1. Clone the github repository.
2. Make a pull request with your `<github-id>_dco.rst` by using this form.
    a. If you wish to maintain anonymity, you can email us that document instead.

3. Along with your code contribution, make sure you:
   a. Create the associated tests in the tests directory.
   b. Include docstrings for the new modules/classes/functions and the functionality is fully described in the documentation.
4. Follow the contributing instructions and make sure you include the "Covered by <dco>" line in the commits.
5. Make a pull request.

For any questions regarding the process of contributing code, please email `hobbes-dev@morganstanley.com`.