Symmetry language extensions for Go

Michael Stay (stay@pyrofex.net)

Jun 10, 2022

Summary

The Silvermint platform needs a language for writing smart contracts. Other platforms have suffered attacks involving the loss of hundreds of millions of dollars worth of tokens due to bugs in contracts. To help avoid this, we want to give programmers the ability to formally verify their code. On the other hand, we also want to avoid having to develop an entire ecosystem — compiler, libraries, editors, debuggers, package management, etc. — around a new language. There is the practical fact that the top 40 languages, with the exception of Prolog at #20 and Haskell at #38, are eager, imperative programming languages. We need interactions with smart contracts not to interfere with the Casanova consensus algorithm that's at the heart of Silvermint's performance. And finally, we want programmers to be able to use Symmetry to write both the on-chain contracts and the off-chain servers that interact with them using the same language.

We decided to use an existing mature imperative programming language and then add features to it to support formal verification. We chose Go, a relatively simple, memory safe, strictly-typed language in the C family. With the exception of unsafe operations, our language, Symmetry, is a strict superset of Go, so nearly every Go program is a Symmetry program. Because we are developing in Go, once Symmetry is mature enough we will also start using the extensions to formally verify our own code, not just smart contracts written by users.

To formally verify programs, we are using Microsoft's Z3 tool, a satisfiability modulo theories (SMT) solver. Z3 is the power behind the Dafny and ZetZ languages, formally verifiable languages from which we're borrowing many features.

We also introduce "layer" keywords that allow programmers to restrict portions of their code to interesting subsets of the language that are easier to verify. For example, one layer keyword restricts Go to a side-effect-free subset of the language; code defined in that language cannot cause side effects except through the objects it receives as parameters.

To run contracts in parallel, Silvermint contracts use the communicating event loop model familiar to all web developers. When a contract processes a message, it begins a transaction; only after the message processing is complete and the new state has been committed are the outgoing messages put in other contracts' message queues. Some contracts need to communicate with other contracts within a single transaction, like a flash loan for a collateral swap. These contracts can enter into a "distributed transaction" that can roll back the effects across multiple contracts over multiple blocks if an error occurs.

Finally, Symmetry is an object-capability-secure (ocaps) language. Functional programming languages allow programmers to reason about side effects by banning them entirely. Ocaps languages, on the other hand, permit side effects, but in a principled way: code can only cause side effects by invoking methods on objects within their lexical scope. Unsafe operations (e.g. from the unsafe package or the reflect package) are forbidden.

Algebraic Data Types

One feature of most functional programming languages that we missed in Go was algebraic data types. Symmetry introduces a new keyword sym_`sum`. The following code

```
sym_sum Tree[X any] {
      type Node struct {
            Data X
            Left Tree[X]
            Right Tree[X]
      }
      type Leaf struct {}
}
```

transpiles to something like

```
type Tree[X any] interface {
    is628940432577(X)
}

type Node[X any] struct {
    Data  X
    Left  Tree[X]
    Right Tree[X]
}
```

```
func (Node[X]) is628940432577(_ X) {}

type Leaf[X any] struct{}

func (Leaf[X]) is628940432577(_ X) {}
```

A `sum` denotes an interface with a private, randomly-named method implemented by the contained structs. Since it's private and unpredictable, no other structs can implement it, even if they're declared in the same file. This implies that the set of structs that do implement it is known at compile time. When doing `switch t := tree.(type) { … }`, the type checker will emit an error if all cases aren't handled.

ADTs will also come with built-in `transform` and `reduce` methods. These are often called `map` and `fold`, respectively, in other languages. We couldn't use "map" because it's already a keyword in Go, so we chose to use the C++ terminology.

Advanced type system and Z3

Z3 is a Satisfiability Modulo Theories (SMT) solver from Microsoft. Many languages use Z3 for proving properties of code. We'll have some combination of Dafny and ZetZ features for formal verification, including preconditions, postconditions, inline assertions, and proof obligations in interfaces.

Go 1.18 introduced generics based on the Hindley–Milner type system. These are "intrinsic" or "Church" types. We can use Z3 assertions to augment the type system with "extrinsic" or "Curry" types[1]. For example, the Document Object Model used in all web browsers includes the `addEventListener` API, which takes a string as its first parameter and an event handler function as its second. The kind of events sent to the handler depends on the string. Given a similar interface in Symmetry, a Z3 assertion would prevent `addEventListener` from being called with mismatched parameters. This is an instance of a sigma type, a kind of dependent type.

To get the full strength of Z3, we'll essentially need to write a compiler from Go into Z3. This will be a long process, so we'll be providing incremental improvements to coverage of the language as we release new versions.

[1] For a comparison of intrinsic and extrinsic types, see, e.g. Functors are Type Refinement Systems.

Determinism and on-chain execution

Symmetry is intended to run on the blockchain, where many different computers have to agree on the computation. We have three options regarding nondeterminism on the blockchain:

1. implement our own deterministic scheduler
2. record message arrival order for all messages so that the other validators can check that such a message is allowed to be delivered and deliver them in that order
3. remove channels from the language on chain

Option 1 means there is only one order in which otherwise parallel processes could run. Option 2 allows more freedom for the execution order, but requires sending traces as part of the information in the block, which bloats blocks and slows down the network. Option 3 is vastly simpler to implement, but means not all Go programs are Symmetry programs. We will probably start with option 3 and then change to 1 later.

Go does not specify the order in which iteration over a map should occur. Symmetry's ADTs will have `transform` and `reduce` methods that are implemented in the Core layer and require Core functions as parameters. Symmetry will also provide similar purely functional implementations for slices and maps. On slices, the function provided to the `reduce` method must be associative; on maps, it must also be commutative. These constraints mean the result is independent of the iteration order.

When using the `for-range` construction, however, executing on the blockchain requires that we either choose an order or verify that the order given is one that could actually occur. Rather than make the programmer think about all possible orders and the implications, we will choose a deterministic order for maps. One might think that keys could be sorted, but not all key types support sorting. Insertion order is a reasonable alternative that works independent of the key type.

Layers

Symmetry uses "layers" to restrict language features for security purposes. The principal layers are a side-effect-free layer, a single-threaded layer, and the full language, which includes all of Go as a subset.

Purely functional languages like Haskell allow programmers to write stateful code by using a monad. Dually, we can enforce side-effect-free code in an imperative language by using a comonad. Symmetry provides three comonads that enforce the language constraints on the lower three layers.

If we take option 1 for nondeterminism, **stateful** and **full** below are distinct; if option 3, they're the same. With option 3, objects within a contract could invoke methods of other objects in the

same contract either synchronously or asynchronously, but not in parallel (except as noted below for transform/reduce or side-effect-free code like core).

- **core**: no mutation, only for-range loops + ADTs + proof types in interfaces. Turing complete. Note that core is not purely functional. Given a typical getter/setter pair for a value, the getter would be in core and the setter in stateful. Because the getter could return different values at different times, it does not behave exactly like a pure function even though it cannot cause side effects
- **stateful**: core + rest of single-threaded Go + parallel transform/reduce on code that's provably commutative & associative
- **full**: stateful + channels + goroutines (for blockchain, we would need deterministic threading)

We have also defined these other sublanguages, but it's not clear yet where or how they'd be used:

- **data**: pure data, "GOON" Go Object Notation, with strings, bigints (as syntax, may not fit in int when parsed), floats (may lose info when parsed, e.g. numbers bigger than 1e+404 may parse as infinite), structs, arrays, maps, nil.
- **expression**: pure data + field lookup + array indexing + arithmetic operators.
- **functional**: core with the added restriction that functions may not close over variables that are also closed over by stateful or full functions

Symmetries

All comments with the exception of "distributed transactional" apply to the layer on that line and those above it.

data	no computation
expressions	can be evaluated in linear time if fixed numeric (int, float) sizes are used, polynomial if bignums are used
functional	pure functional
core	no side effects
stateful	single threaded except for parallelization in transform/reduce with functions proven commutative, associative
full	locally transactional; deterministic on chain if

	we choose option 1, replicable if 2

Auditors

The type checker checks each layer with a predicate on an (AST, symbol table) pair called an "auditor". Quoting from Yee and Miller:

> We can get other kinds of constraints when the auditor does inspect the code. Here are some examples of interesting semantic properties:
>
> - **confined**: the object does not overtly transmit outward any information or authority it receives in messages (it can only transmit information or authority with which it was endowed upon creation)
>
> - **deterministic**: the object cannot obtain any information or authority except through messages sent directly to it (that is, a log of the received messages is sufficient to replay its entire behaviour)
>
> - **functional**: every method on the object has no side effects and produces an immutable result depending solely on its arguments
>
> - **frozen**: the object does not ever mutate any local bindings (or in other words, it is indistinguishable from a duplicate that contains copies of the bindings for the free variables accessed by the object's methods)
>
> - **deep frozen**: the object cannot ever cause the mutation of anything
>
> - **open source**: the object provides access to all of its source code through a standard interface
>
> - **open state**: the object provides access to all of its internal state through a standard interface
>
> - **transparent**: the object is both **open source** and **open state**
>
> - **pass-by-copy**: the object is safe to transmit to another party by sending a copy of the object's source code and state
>
> Although determining that a program will *actually* behave in a particular way is undecidable in general, for all but the last of these properties we can define a straightforward check that admits a reasonable subset of the space of acceptable programs in practice.

We intend to allow users to define their own auditors and layers.

Contracts

A contract is an object server. Messages to the server contain a reference to an object in the server's memory, a method name, a list of arguments, and a location to send the result of the invocation. Usually, the server runs the method code synchronously, providing promises for the result of invocations of methods on objects in other contracts. If the code completes successfully, the messages get added to the other contracts' message queues.

Unless proven otherwise, the order in which messages get delivered to a contract matters. Therefore, the Silvermint network decides on an order for the messages. Each message sent to a contract must specify an epoch. Validators keep track of the current epoch of each contract, and increment it whenever they see a message to that contract. This means that the number of validators is an upper bound on the total number of messages that can contend for access to a contract during any given epoch.

Usually the network will sort the messages in an epoch by hash, and deliver them one by one. However, some contracts may prefer to order them by some other criterion; for example, an order book would want to sort bids and asks by price. Each contract decides how it wants to receive messages.

When messages depend on each other, we will deliver them in E-Order, the closest we can get to causal order using only cryptography.

Certain applications, like flash loans for collateral swap, require multiple contracts to interact within a single transaction. There are existing protocols for supporting distributed transactions, but we haven't yet pinned down how Symmetry will provide this ability. We may add special syntax or we may provide a library.

Conclusion

The Symmetry design provides a formally verifiable, ocaps-secure, eager, imperative, mature programming language with an opt-in side-effect-free sublanguage. It also encompasses all of existing Go, which provides a rich ecosystem for developers. We anticipate that these features will make Symmetry an attractive choice for new blockchain programmers as well as for systems engineers concerned about the reliability of their code.