Short Paper: Blockcheck the Typechain

Downloaded from: https://research.chalmers.se, 2023-10-26 15:49 UTC

Citation for the original published paper (version of record):

N.B. When citing this work, cite the original published paper.
Abstract
Recent efforts have sought to design new smart contract programming languages that make writing blockchain programs safer. But programs on the blockchain are beholden only to the safety properties enforced by the blockchain itself: even the strictest language-only properties can be rendered moot on a language-obliviou blockchain due to inter-contract interactions. Consequently, while safer languages are a necessity, fully realizing their benefits necessitates a language-aware redesign of the blockchain itself.

To this end, we propose that the blockchain be viewed as a typechain: a chain of typed programs — not arbitrary blocks — that are included iff they typecheck against the existing chain. Reaching consensus, or blockchecking, validates typechecking in a byzantine fault-tolerant manner. Safety properties traditionally enforced by a runtime are instead enforced by a type system with the aim of statically capturing smart contract correctness.

To provide a robust level of safety, we contend that a typechain must minimally guarantee (1) asset linearity and liveness, (2) physical resource availability, including CPU and memory, (3) exceptionless execution, or no early termination, (4) protocol conformance, or adherence to some state machine, and (5) inter-contract safety, including reentrancy safety. Despite their exacting nature, typechains are extensible, allowing for rich libraries that extend the set of verified properties. We expand on typechain properties and present examples of real-world bugs they prevent.

1 Introduction
Smart contracts are rife with bugs. Invariably, these bugs result in the loss of dollars, at times measured in the hundreds of millions. Many of these bugs can be traced back to design decisions, inconsistencies, and footguns in the contract’s programming language [3, 13, 26, 31, 45]. Evidently, we should improve our smart contract programming facilities.

Recent efforts in industry and academia have sought to design new, strongly typed smart contract languages to assuage these concerns. While a step forward, we maintain that these efforts include or lack key language properties that will inevitably lead to similar bugs. For example, Flint [34], Scilla [35], and Obsidian [8] abort execution due to runtime faults, a design decision that can lead to contracts that inadvertently lock funds indefinitely [11, 30, 39]. Neither Bamboo [44] nor Vyper [40] enforce asset linearity, allowing for asset loss or duplication as well as inconsistency between the accounting and presence of value, two common pitfalls [3, 11, 12]. And while Nomos [10] excels at avoiding these prior deficiencies, it limits inter-contract interactions and is susceptible to deadlocks when they do occur.

Consequently, we hold that smart contract programming languages necessitate a type system with properties that specifically capture smart contract correctness. In particular, typechecking in such a language should minimally verify:

1. **asset linearity and liveness**, to prevent the loss, duplication, and inaccessibility of value
2. **physical resource availability**, including CPU and memory, to ensure the possibility of executing to completion
3. **exceptionless execution**, or lack of unchecked early termination, to enforce complete error handling
4. **protocol conformance**, or adherence to some state machine, to prevent executing code in unintended states
5. **inter-contract safety**, to prevent reentrancy bugs [12] while allowing safe inter-contract interactions

But a language-only approach is insufficient: failure to consider inter-contract interactions precludes verification of these desired properties. For example, to verify a contract exception-free, every contract it interacts with must be exception-free. Similarly, for any contract to soundly treat an asset linearly, all contracts must treat that asset linearly. Thus, realizing strict language properties on the blockchain requires the blockchain itself to be language-aware.

To this end, we propose a type-driven, language-aware redesign of the blockchain [29, 42] as a typechain: a chain of typed contracts, state, and transactions — not arbitrary blocks — that are accepted iff they typecheck against the existing chain. The typechain is blockchecked by miners to validate typechecking, verifying, at minimum, the aforementioned properties.
While typechains have stronger properties than many strongly typed languages, they may not sufficiently capture all desired smart-contract correctness properties. Thus, a typechain must be extensible, allowing for rich libraries that extend its set of verified properties. For example, whereas prior work has developed entirely new blockchains to enable on-chain privacy [5, 7, 9, 16, 20] or verified off-chain computation [2, 4, 18, 22, 24, 27], a typechain allows this functionality to be implemented as a library without modification to its core protocols or language.

In the following section, we expand on typechains, detailing their desired properties, and present examples of real-world bugs they preclude (§2). Following, we illustrate how libraries can extend a typechain’s set of verified properties (§3). Finally, we provide concluding remarks (§4).

2 Formulating the Typechain

A typechain is an extensible sequence of statically typed programs ordered and grouped into blocks. To add a block to the chain, it must be blockchecked: a distributed network of machines must agree that the program formed by adjoining the new block is well-typed.

Formally speaking, a program \( p = \bar{d}, s \) on the typechain is a sequence of typed declarations \( \bar{d} \) and closed statements \( s \). The typechain is a sequence of programs \(^1\) \( C_T = \bar{p} \) where

\[
\forall p \in C_T, \text{blockcheck}(p) \iff \text{blockcheck}(C_T).
\]

Newly chained programs — contracts or transactions — must typecheck individually, and the program formed by the concatenation of existing programs and the new program must also typecheck. Thus, successfully blockchecking a new program \( p \) can be seen as swapping some existing typechain \( C_T \) for a new typechain \( C'_T \) where

\[
C'_T = C_T; p \quad \text{if} \quad \text{blockcheck}(C'_T).
\]

From a different yet isomorphic perspective, a typechain is a refinement (by one account) or restriction (by another) of a blockchain in which miners’ core task is to reach consensus on the type safety of the program resulting from the concatenation of all programs on the chain. All safety properties, which traditionally arise from the blockchain’s runtime [42], arise instead from the type system. This includes account accounting, proof of ownership of assets, and the ability, financial or otherwise, to compute or store on the blockchain. The deployment of a contract or execution of a transaction is pre-conditioned on these properties.

A typechain’s type system is thus stronger than that of many conventional strongly typed languages. Minimally, it must guarantee the properties enumerated in §1. We emphasize that, while helpful, a subset of these properties is not sufficient; a single omission is enough to admit the very class of nefarious, money-squandering bugs we wish to avoid. Conversely, this minimal set of properties is necessarily insufficient to capture all desirable correctness properties, so typechains are extensible, allowing for new, arbitrary properties to be admitted and enforced (§3).

In the following sections, we detail the properties in §1, providing examples of bugs that each would prevent. Our examples are written in pseudocode resembling Rust [33, 36].

\(^1\)Note that \( C_T = \bar{p} = \bar{d}, s = p \), so the typechain is itself a program.

1
impl Auction when open {
2 fn bid(self, amount: Money) {
3 if self.should_close() {
4 close self;
5 } else if amount <= self.max_bid {
6 amount.return_to_sender();
7 } else {
8 self.max_bid = amount;
9 }
10 }

Figure 1: A buggy Auction contract with a bid() method.

Asset Linearity and Liveness

A blockchain is primarily concerned with managing and operating on assets: currency, arbitrary items, tokens (sub-currencies), and so on. As in the real world, correctness demands that assets are never lost nor duplicated.

Unfortunately, this property is difficult to maintain in existing blockchain languages [3, 12, 13], where assets are trivially reusable and dispensable. As an example of what can go wrong, consider the bid() method in Figure 1. The intent is that, while the auction is open, bids of amount can be placed by invoking Auction::bid(amount). The bidder becomes the highest bidder if their bid is the highest thus far. Otherwise, the bid is returned. Can you spot the problem(s)?

The first branch, which closes the auction when necessary, fails to return the unconsidered bid amount to its bidder. The third branch, which runs if amount is the highest bid, overwrites max_bid without returning this previous highest bid to its bidder. In both cases, value is irreversibly lost.

A fix is to call the appropriate return_to_sender() in each branch. A language with linear assets would statically enforce the existence of such calls. Linear types [17, 41] provide a mechanism: encode assets as linear types, enforcing their use exactly once, preventing loss or duplication.

However, linearity alone is insufficient: typechains must also enjoy asset liveness, or the guarantee that contracts are always in a state that allows their assets to be transferred. Failure to enjoy asset liveness can result in assets that are locked and rendered unusable. One famous instance of such a bug led to the loss of ~200 million USD [32], though many other instances have occurred in the wild [3, 11, 13].

Physical Resource Availability

To protect miners from executing runaway transactions, themselves arbitrary programs, blockchains enforce limits on computational resources including the CPU and memory. Overstepping transactions are halted or partially reverted.

This behavior can lead to unexpected consequences, especially when state is only partially reverted [11, 23, 39]. As an example, we consider again bid() in Figure 1. In languages like Solidity [15], send calls like return_to_sender() can fail if such a call would exhaust the stack limit or if other allotted computational resources would not suffice. If the call fails, a false value is returned [3, 15]. Unfortunately, contracts often ignore this value, leading to a class...
of bugs known as unchecked send [3, 45]. An unchecked send in the `bid()` method, for example, would result in the `Auction` contract inadvertently holding more funds than desired.

Contracts may also settle in a state where execution without exhausting resources, and thus execution at all, is impossible [30]. Such a contract has its funds locked forever. This entails that asset liveness necessitates resource availability. As an example, Government-Real’s only asset extraction method iterated over an array that grew too large to scan within the allotted limit; its funds were locked without recourse [39].

A typechain statically guarantees the availability of physical resources for all possible executions of a contract. Mechanically, its type system produces concrete, input-independent resource usage upper bounds. Programs exceeding limits fail to blockcheck. While languages like Nomos [10] and Scilla [35] demonstrate feasibility of such a guarantee, a challenge remains in finding mechanisms that are developer-friendly.

Exceptionless Execution

In addition to resource exhaustion, early termination in existing blockchains can arise due to arbitrary unchecked exceptions [25, 38]. In Solidity, contracts can arbitrarily throw exceptions, and in Solidity, Flint, and Obsidian, exceptions can arise from incorrect logical operations such as out-of-bounds indexing and division by zero. While their sources of early termination differ, unchecked exceptions and resource exhaustion admit similar issues. Thus, a typechain must either guarantee that all possible exceptions are checked or prohibit exceptional operations entirely, perhaps by leveraging dependent types [43] or SMT solvers [1].

Protocol Conformance

Smart contracts typically follow a protocol: a predefined series of steps conditioned on the contract’s state. For instance, `Auction` in Figure 1 allows a bid only when it is open.

Existing blockchains require programmers to enforce protocols in an ad-hoc, unstructured, and dynamic manner [14]. Often, they advocate for throwing exceptions when expected conditions fail to hold, violating exceptionless execution. Of course, mistakes abound in practice [13, 25, 38]. Typechains, instead, allow programmers to directly encode a contract’s protocol; conformance to the protocol is then statically enforced by the type system.

Recent efforts have recognized the importance of protocol conformance: Obsidian [8], Flint [34], and Nomos [10] all statically enforce protocols, the former two by leveraging typestate [37] and the latter by leveraging session types [19].

Inter-Contract Safety

Blockchains like Ethereum allow smart contracts to interact through mutual method invocation. While merely innocuous, interaction can lead to vulnerabilities in both the caller and callee. For instance, given a stack frame limit of \( k \) frames\(^2\), a caller can coerce a callee into exhausting its frame limit by making \( k - 2 \) nested calls prior to invoking the target callee. When the callee makes a call of its own, it will exceed the frame limit and an exception will be thrown. As discussed, if unexpected, such exceptions can lead to vulnerabilities.

\(^2\)In Ethereum, \( k = 1024 \).

Callers are inherently vulnerable. Since a callee can react arbitrarily, it may re-invoke the caller. If the caller expected logically atomic execution, its re-execution may result in inconsistent state which in turn may lead to vulnerabilities. Such vulnerabilities are termed reentrancy vulnerabilities.

The issue is further exacerbated by the ability for contracts to interact unknowingly. In Ethereum, for instance, asset-transfer calls, such as `return_to_sender()` in Figure 1, invoke a method on the recipient’s contract. Infamously, ~40 million USD was stolen from TheDAO via a reentrancy attack resulting from such a call [12]. The attack was considered so unacceptable that Ethereum forked, mutating the purportedly immutable blockchain in order to make the stolen funds recoverable [28]. The particularly vile nature of reentrancy vulnerabilities has led to the development of myriad tools and methods to detect them [6, 21, 25, 30, 38]. These approaches require manual application by the programmer, and to date, none is both sound and complete.

A typechain enables safe inter-contract communication by guaranteeing that local contract reasoning extends globally. This is facilitated by the properties enumerated thus far. For example, exceptionless execution and resource availability immediately obviate caller-originated resource exhaustion attacks, while asset linearity ensures that inter-contract interactions, even reentrant ones, cannot render asset accounting logically inconsistent, the crux of TheDAO attack. In addition, a typechain’s type system should be statically aware of all inter-contract interactions and conservatively forbid reentrant calls, perhaps by leveraging session types [19] or communicating automata [35].

3 Extending the Typechain

A typechain’s properties are necessary but insufficient: a smart contract is correct only if these properties hold, but desired correctness properties may be absent.

To compensate, typechains are malleable: a typechain’s type system is extensible (§3.1) and allows for rich libraries that leverage Curry-Howard isomorphisms to expose types that act as proof witnesses (§3.2). Thus, a typechain’s type system must at once be strict enough to allow enforcing arbitrary correctness properties while permissive enough to host expressible libraries.

3.1 Type System Strengthening

Smart contracts can extend the set of properties verified by a typechain via extensions that strengthen the typechain’s type system. Extensions run during blockchecking and apply to the contract that deployed the extension as well as its dependents. Properties checked by extensions are guaranteed to hold uniformly across the entire typechain. Any property that can be expressed as a finite static analysis can be implemented as an extension.

Concretely, extensions are type systems that are required to be strengthenings of the typechain’s type system. We say that a type system defined by the relation \( \Gamma \vdash e : \tau \) is a strengthening of a type system defined by \( \Gamma' \vdash e' : \tau \iff \Gamma' \vdash e' : \tau \implies \Gamma \vdash e : \tau \).

We call \( \Gamma' \vdash e : \tau \) the strengthened type system; \( \Gamma \vdash e : \tau \) is the original type system. A strengthened type system admits monotonically fewer programs than the original. In other words, it reduces
the set of programs that typecheck. Critically, a strengthening cannot weaken a type system: a property that holds in the original must hold after strengthening. Consequently, extensions cannot deliberately or accidentally weaken a typechain’s properties.

Example: Global Singleton

As an example of a property verifiable via extensions, consider the global singleton: a type with at most one instance in a program. We would like to implement a type, Singleton, external to the typechain, that is guaranteed to be a global singleton in the typechain. In other words, we would like to verify that at most one instance of type Singleton is ever constructed in the typechain.

To begin, the Singleton type is declared as the usual container generic over an embedded type T:

```plaintext
struct Singleton<T> { }
```

We define a flow-sensitive strengthening of \( \Gamma \vdash e : \tau \) as

\[
\begin{align*}
\Gamma \vdash e : \tau &\quad \text{singleton}(\tau) \quad 0 \\
\Gamma, \Delta \vdash \alpha : \tau &\quad \Rightarrow \quad \Delta' \\
\Gamma, \Delta \vdash \alpha : \tau &\quad \Rightarrow \quad \Delta' \cup \{ \tau \} 
\end{align*}
\]

where \( \text{singleton}(\tau) \) is true when \( \tau = \text{Singleton}::<T> \) for some concrete type \( C \), and \( v \) is a value. Rule 0, to the left, ensures that all non-Singleton types typecheck as usual. Rule 1, to the right, asserts that a Singleton instance can only arise from one value in the program. Assuming instances must be constructed as values \( v \), this extension guarantees the global singleton property we sought.

3.2 Type Libraries

While many existing type systems are readily leveraged to codify correctness properties, the practice is notably absent in blockchain programming, owing to weaker type systems. Typechains nullify this dichotomy by being flexible yet strict enough to allow such properties to be codified in libraries.

Example: Timing

We consider time-based properties as an example, where correctness is specified as a function of time. A typechain should allow the implementation of a library contract type \( \text{Timed<Auction>} \), generic over a contract \( C \), that automatically and indelibly enforces centrally declared, time-based properties. Leveraging such a library for Auction in Figure 1 might resemble:

```plaintext
impl Timing for Auction {
    fn check(self, time: Time) {
        if self.should_close(time) close self;
    }
}
```

This Timing implementation specifies time-based properties for Auction: if an auction should_close(), it is, in fact, closed. By deploying a \( \text{Timed<Auction>} \) in place of an Auction, the library would automatically enforce this property. Consequently, Auction can be written without further calls to self.should_close(). In particular, methods implemented on open contracts, such as Auction::bid(), can rely on being called only when the auction is logically open without further confirmation.

4 Concluding Remarks

A typechain is a sequence of typed programs blockchecked by a distributed network of machines that verify their well-typedness. When the type system ensures key safety properties, bugs plaguing smart contracts cease to be. These properties must minimally include (1) asset linearity and liveness, (2) physical resource availability, (3) exceptionless execution, (4) protocol conformance, and (5) inter-contract safety. Existing work does not go far enough, enjoying only a subset of these properties. Unfortunately, a single omission permits the very bugs we attempt to evade.

Verification of these properties must holistically consider all programs on the chain: a single errant contract thwarts soundness. Thus, unlike verification applied to individual contracts, typechains enjoin safety properties universally, requiring them to hold of all contracts. Still, a typechain may not sufficiently capture all desired correctness properties, so typechains are extensible, allowing user-defined properties to be verified during blockchecking. By blockchecking safety and user-defined correctness properties, the ambit of verification is approached.

We believe typechains stand to fundamentally improve the safety and correctness of smart contracts and are actively working on the design and implementation of a typechain.

Acknowledgments

Thanks to Dan Boneh, Fraser Brown, and the anonymous reviewers for their invaluable feedback and suggestions.

References


