CHALMERS
UNIVERSITY OF TECHNOLOGY

Safer smart contracts through type-driven development

Using dependent and polymorphic types for safer development of smart contracts

Master's thesis in Computer Science

Jack Pettersson and Robert Edström

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Safer smart contracts through type-driven development

Using dependent and polymorphic types for safer development of smart contracts

ROBERT EDSTRÖM, JACK PETTERSSON

Department of Computer Science and Engineering
Computing Science Division
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2016

Safer smart contracts through type-driven development
Using dependent and polymorphic types for safer development of smart contracts
ROBERT EDSTRÖM, JACK PETTERSSON

Supervisor: K.V.S. Prasad, Department of Computer Science and Engineering
Examiner: Gerardo Schneider, Department of Computer Science and Engineering

Department of Computer Science and Technology
Computing Science division
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2016

Safer smart contracts through type-driven development
Using dependent and polymorphic types for safer development of smart contracts
ROBERT EDSTRÖM, JACK PETTERSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We show how dependent and polymorphic types can make smart contract development safer. This is demonstrated by using the functional language Idris to describe smart contracts on the Ethereum platform. In particular, we show how one class of common errors can be captured at compile time using dependent types and algebraic side effects. We also bring type annotations to the realm of smart contracts, helping developers to circumvent another class of common errors. To demonstrate the feasibility of our solutions, we have extended the Idris compiler with a backend for the Ethereum Virtual Machine. While we find that the functional paradigm might not be the most suitable for the domain, our approach solves the identified problems and provides advantages over the languages in current use.

Keywords: smart contracts, dependent types, polymorphic types, Ethereum, blockchain, Idris, functional programming

Acknowledgements

We would like to send gratitude to our supervisor K.V.S. Prasad, who has been very supportive and helpful with setting the direction of our work. Additionally, the vibrant Ethereum and Idris communities have both been very responsive and helpful—in particular, the members of the `#idris` IRC channel. Also to Kieren James-Lubin who helped provide the initial idea for this project, as well as Joseph Lubin and the others at ConsenSys for bringing us to DEVCON1. Finally, the various people who have helped us with feedback and guidance during our work.

Robert Edström and Jack Pettersson, Gothenburg, February 2016

Intended audience

This report is aimed at readers with an undergraduate in computer science or similar. In particular, familiarity with a typed functional language like Haskell, F# or a member of the ML family will be very helpful for reading code samples. Some familiarity with programming language implementation is expected as well. Public-key cryptography will be briefly discussed, but fully understanding these parts is not vital to understand our contribution.

In November 2015, representatives from both academia and the industry with an interest in blockchain and consensus technology got together in London, at Ethereum's first annual developer's conference, DEVCON1. We gave a short presentation on this thesis and our project there. A recording of our talk is available online at `https://www.youtube.com/watch?v=H2uwUdzVD9I`. If the reader is not familiar with smart contracts in general and Ethereum in particular, we recommend that our presentation is watched after reading chapter 1 and section 2.1. Otherwise, we believe that the presentation serves well as an intuitive introduction to our work.

Contents

1 Introduction

Contracts—conditional agreements between two or more parties for doing or not doing something specified—are a vital underpinning of modern finance, business and law. As business and finance are increasingly automated, the incentives to let computer programs interpret, enforce and execute contracts also increase. Naturally, it is crucial that the interpretation of such contracts is unambiguous and deterministic.

It has been shown that contractual agreements are very well suited to be expressed in formal languages [1]. Most contractual clauses are on the form "given x, allow/perform y", where x is an event or point in time and y is an action. Also note that both x and y may be referring to, or be composed of, other contractual clauses. Consider for example a simple contract that grants its holder the right to buy or sell a particular asset at a fixed price when the contract expires. This could easily be represented by a simple program, as seen in fig. 1.1. The details of this program are not important, what is important is that contracts essentially consist of mere logic, despite the extensive jargon used in both the legal and financial fields. This observation is not new and there have already been several efforts to design formal languages and programming libraries intended to analyze or execute financial contracts [1]–[4].

```
function buy() {
  if(currentTime == expiryTime) {
    seller.give(buyer, asset)
    buyer.give(seller, price)
  }
}
```

Figure 1.1: Object-oriented pseudo-code encoding the contract "At `expiryTime`, the `buyer` may buy the `asset` from the `seller` at the specified `price`". The identifier `currentTime` is an environment variable reflecting the time of execution.

However, all of the existing implementations suffer from the same fundamental problem: if a contract is executable, the involved parties have to trust the executor to give them the correct result. It would be desirable to have the code "execute itself", without any possibility of interference from an executor or other third parties. It turns out that this has recently become both possible and practical, thanks to programmable *smart contract* platforms.

1.1 Smart contracts

The term “smart contract” is a bit vague, but we will take it to mean programs or protocols that encode and enforce agreements [5]. Some examples of smart contracts are digital rights management schemes for multimedia files and admission control schemes in computer networking. One could also view vending machines as a primitive precursor of smart contracts, enforcing the agreement that a coin can be traded for e.g. a can of soda [6].

It should be noted that smart contracts are not merely “executable contracts”. Contracts are traditionally external to the system they regulate and are prescriptive rather than descriptive. They don’t define what is *possible*, but what is *permissible* and the consequences of breaching the prescriptions. Conversely, smart contracts define *how* entities may transact, and automatically execute these transactions when asked to. The distinguishing feature is that once initiated, the participating entities cannot reverse or stop the transaction unless allowed by this or another smart contract; the agreement is *enforced*. As a consequence of this, smart contracts cannot be said to be violable in the same way as traditional contracts. If a smart contract is invoked it is automatically honored, otherwise it does not apply. Nevertheless, a greater degree of automation should be possible if traditional contracts can be complemented—or partially replaced—by smart contracts, especially in the financial and legal fields.

The previously mentioned examples of smart contracts are very specialized. In recent years however, programmable smart contract platforms have emerged. They offer developers programming languages to specify arbitrary smart contracts in, which when deployed to the respective platform are guaranteed to be executed exactly as specified [7]. Since smart contracts define the possible transactions rather than the allowed behavior, smart contracts on these platforms generally resemble traditional computer programs. The platforms are completely agnostic to the purposes of the programs they execute, hence it is possible that some of them are not contracts in any meaningful sense. Thus, it could be argued that referring to all programs on these platforms as “smart contracts” is a misnomer. Nevertheless, this is the jargon of the field, and it does seem like smart contracts is the main use case of platforms which enforce arbitrary rules. Programs on these platforms are said to be *trustless*, because users needn’t trust any third party when using them; what needs to be trusted is just the code itself [8].

Smart contract platforms enable the implementation of applications such as transparent crowdfunding platforms without middleman fees [9], automated organizational governance tools [10] and censorship-resistant prediction markets [11]. They also seem to be fit for use in the emerging field of the Internet of things. IBM has carried out promising experiments with using a smart contract platform as the decision making backbone for connected devices all over the world [12], [13]. Currently, the Internet of things is based on centralized servers which devices continuously send data to. The servers aggregate data from numerous devices to decide parts of the devices’ behavior. Disregarding the obvious drawbacks with a single point of failure, this architecture has two major problems. First and foremost, it will not scale to the needs anticipated, since there will soon simply be too many devices. Second, data generated by devices quickly becomes irrelevant; a round-trip to the server—including server-side processing—wastes data that could have been used to provide a better service. Instead, smart contract platforms seem to be able to support a “device democracy”, allowing devices to collectively decide on their behavior without relying on any servers [12].

1.2 Smart contract languages

These developments are certainly very exciting, but if smart contract platforms should become the foundation of our economy—possibly even our technological infrastructure—it is of utmost importance that the technologies we use to develop smart contracts are robust. In particular, smart contract languages should be as safe as possible: errors should be hard to introduce, easy to detect and easy to fix. Unfortunately, this is not yet the case. All languages in active use today are imperative, and it has been shown that they are prone to programming errors due to unforeseen states [14].

Conversely, most of the existing formal languages for financial contracts are functional in nature [2]–[4]. Many of them are also compositional, such that contracts are created by combining smaller contracts. In general, the functional programming paradigm is very well-suited for implementing domain specific languages as combinator libraries, which accommodates such compositionality [15]. Furthermore, purely functional languages are very well suited for testing and formal program verification [16], a very attractive feature for critical domains such as smart contracts. We also note that extensive type systems are generally very useful for catching errors early in program development [17]. As such, we see huge potential in a functional smart contract language with an extensive type system.

1.3 Aim

Our thesis is that a functional language with an extensive type system can allow safer development of smart contracts. Specifically, we consider three classes of errors that are common due to the characteristics of current smart contract platforms. All of these will be explained in depth in section 2.2. If our solutions can help detect two of these problems earlier in development or make it easier to circumvent them, we will consider our thesis proven.

In order to demonstrate the feasibility of what we suggest, we will also provide a proof-of-concept implementation of our solutions, targeting the popular smart contract platform *Ethereum* [18]. The goal is for it to be able to compile simple smart contracts to run on the platform.

1.4 Delimitations

We do not intend to develop a new library or domain specific language specialized on describing specific types of contracts. As mentioned, there already exist several of those and providing yet another one doesn't seem like an important contribution. However, keeping in mind that many of the mentioned libraries are functional, we do hope that a functional language on a smart contract platform will facilitate the implementation of the existing ones in a trustless environment.

The focus is on safety: ease of use is not a main goal. We view this as a later concern, while safety is more fundamental and harder to add after the initial design. It is also outside the scope of this thesis to deliver a complete, optimized or readily usable language implementation.

1.5 Report outline

The rest of this report is structured as follows:

Chapter 2 gives the technical background to our work. This includes Ethereum, the smart contract platform which we use as a testing ground for our ideas. The classes of common errors that were mentioned above are explained in greater detail. The solutions we propose in chapter 3 make use of polymorphic types, dependent types and the programming language *Idris*, all of which will be explained.

Chapter 3 presents our solution, namely how an Idris library that we have designed can be used to detect or circumvent the common errors identified in the previous chapter.

Chapter 4 explains how we have implemented our solutions. Specifically, this includes a proof-of-concept backend for the Idris compiler which targets the Ethereum Virtual Machine (EVM), as well as the library described in the previous chapter.

Chapter 5 evaluates both our theoretical approach and our implementation of it. Advantages, problems and opportunities for future work are identified.

Chapter 6 concludes.

2
Technical background

This chapter will explain the technologies that are relevant to this thesis. It is split roughly into two halves, with the first covering smart contracts and Ethereum, the smart contract platform we are targeting. This will begin with an overview of Ethereum, followed by a more in-depth technical description. After that, problems in the smart contract languages in current use will be explained. The second half covers type systems and Idris, the language we will use to demonstrate our solutions in the next chapter. This will start with a discussion on type systems in general and dependent types in particular, illustrated with examples of Idris code. Finally, we will give an overview of some important features of Idris.

2.1 The Ethereum smart contract platform

In short, Ethereum is a peer-to-peer network in which the participants store and execute programs on an embedded virtual machine. Programs are executed on request by users and may perform any number of actions as a result, such as updating its state, executing other programs or sending value to users or programs. The virtual machine is Turing-complete, so the programs can be arbitrary, but they are generally similar to database transactions. Furthermore, they are often applied to the transfer or registration of digital resources such as property titles, votes, currency, licenses, control over "smart" autonomous devices, stocks and so on. Anyone can execute or deploy programs to the network for a small fee. The participants in the network continuously verify that they agree on the state of the system: in particular, they verify that they agree on the states of the deployed programs. If anyone disagrees, it is easy to verify their claim. Thus, this peer-to-peer network is able to establish unique and verifiable global states according to any rules supplied by its users.

On a conceptual level, programs on Ethereum can be likened to a trusted third party, one which consists solely of code that is distributed over a peer-to-peer network. This conceptual third-party is trusted for correctness, but not confidentiality [14]; the binary code, execution steps and stored data of all programs are completely public. The entire state of the system can be viewed by anyone [18], [19]. On a slightly more technical level, it can be viewed as a shared world computer (albeit very slow) or a cryptographically secured database without administrators [20]. The programs on the platform are executed on request and have a limited running time, rather than running continuously [18], [19]. The mechanics of this will be explained in section 2.1.3.

```
1  contract NameRegistry {
2    mapping (string => string) names;
3
4    function register(string name, string value) returns boolean {
5      if(names[name] == 0) {
6        names[name] = value;
7        return true;
8      }
9      return false;
10   }
11
12   function lookup(string name) returns string {
13     return names[name];
14   }
15 }
```

Figure 2.1: A simple name registration system, allowing users to register and look up names. Line 2 declares an array of strings, indexed by strings. Uninitialized indices map to 0, as seen on line 5. The language used is Solidity[1], a common smart contract language which we will only use for illustrative purposes; it is not necessary to understand the syntax in-depth.

To illustrate how this can be used, we present a simple smart contract in fig. 2.1. It implements a name registration system, similar to a DNS. The implementation is trivial, but note that this program will be executed by a peer-to-peer network that will agree on the currently registered names and their values. Additionally, programs on Ethereum are publicly executable, so anyone with an Internet connection would be able to register a name. In summary: when deployed to Ethereum, this simple program implements a name registration system that is globally available, cannot be controlled or censored, and is essentially free for anyone to use.

We will now try to make this intuitive understanding more general and precise. In short, Ethereum allows any entities—which may be mutually distrustful or even malicious—to arrive at a globally accepted state according to any rules that they previously agreed upon [18], [19]. This is achieved using a distributed algorithm and accompanying replicated data structure that allows a peer-to-peer network to agree on a particular value, even though some of the peers might deviate from the specified protocol [21]–[23]. In the case of Ethereum, the value that the network agrees upon is the current state of the system, which is updated whenever a program is added or executed [18], [19]. For readers familiar with the digital currency Bitcoin, it might help to mention that Ethereum essentially uses the same underlying algorithm, but extends it with a programmable virtual machine to allow the implementation of arbitrary smart contracts, rather than being specialized on currency.

In a more long-term perspective, smart contract platforms like Ethereum might be able to run any kind of application. At the present moment, performance is a limiting factor, so only applications with backend modules that have non-computationally intensive business logic are currently being considered. As previously mentioned, this includes applications such as crowdfunding platforms, organizational governance tools and prediction markets. In general, most applications that include transferring or registration of resources are suitable, especially those that could exhibit a network effect from being connected directly to a global transactional platform.

[1]http://ethereum.github.io/solidity/

2.1.1 Contract structure

In the high-level languages currently used for Ethereum contract development, contracts are structured much like modules or classes in traditional programming languages. Here, a contract consists of a list of data fields and a list of functions which read and modify these, as in fig. 2.1. When we refer to the implementations of smart contracts below, it is useful to keep this in mind. However, contracts are ultimately encoded as byte code for the Ethereum Virtual Machine (EVM) that is embedded in all clients, so their structure can be as diverse as programming languages allow.

2.1.2 Users, contracts and their accounts

Ethereum has two main entities: *users* and *contracts* that send *messages* between each other. If the recipient of a message is a contract, it will execute a bit of code, which might send new messages to other users or contracts. Each message sent by a contract will be part of the same *transaction* that initiated the message. Only users may initiate transactions; contracts are passive procedures run by receiving properly formatted messages [18], [19]. Messages and transactions are not transmitted from one computer to another, but are reflected solely in the system's state [18], [19].

Because of Ethereum's completely open and permissionless architecture, it would not be possible to require users to go through a registration process to use it. Still, some notion of user identities or *user accounts* is required for most use cases, which is achieved using *public-key cryptography* [18], [19]. Public-key cryptography is a widely used cryptographic technology that can be used for both encryption and authentication purposes, but only the latter is relevant here. It is based on the notion of *key pairs*. As the name suggests, a key pair consists of two keys: one *public*—representing an identity—and one *private*—used to prove ownership of the public key [24]. A public key can be thought of as a color—say a certain shade of red—and the private key as a crayon with that color. Given a letter written in this shade of red, it can be assumed that only a person with access to this crayon could have written the letter. The public key can be freely shared with anyone, but the private key has to be kept secret at all times. Since user accounts are directly tied to key pairs, users never register accounts. Instead, a particular user account can be used only with access to its private key [18], [19].

There are also *contract accounts*, each of which is controlled by the program which constitutes the contract. Contract accounts have access to a non-shared persistent memory, which they use to store their state [18], [19].

Each account—i.e. both users and contracts—has exactly one automatically generated *address*[2], which is used as its unique identifier [18], [19]. Additionally, all accounts have an *ether* balance. Ether is a transferable money-like asset that is built into the Ethereum system[3] [18], [19]. While it is possible to use ether as a currency, its purpose is to prevent certain attacks, as will be explained in section 2.1.4.

[2]For the curious reader, the address of a user account is a hash of its public key, while the address of a contract account is, roughly speaking, a hash of the message that created it.

[3]For the readers familiar with Bitcoin, ether is to the Ethereum network as the *currency* bitcoin is to the Bitcoin *network*.

2.1.3 Code execution and transactions

The Ethereum state of the Ethereum system is updated through the propagation of messages as part of transactions. Transactions can be thought of as transactions in a database. All transactions include at least one message—as will be elaborated on shortly—and are always initiated by a *user* account. This message might also result in new messages, which will be part of the same transaction. Additionally, transactions are atomic; if one of the executions triggered by a transaction fail, the transaction has no effect at all [18], [19].

Each message has exactly one receiving account and may contain an arbitrary payload [18], [19]. It is possible to include an amount of ether to transfer with any message. Any ether received by a contract is added to its balance and is controlled exclusively by its code, while ether received by a user is controlled by its private key [18], [19]. A message may do the following things:

Transfer ether The simplest kind of message is to only transfer a certain amount of ether from one account to another, but ether may be included in more complex messages as well. It is worth reiterating that in general, Ethereum transactions are more similar to database transactions than monetary ones. To transfer value between users is simply one particular way to update the state.

Contract execution If a contract receives a message, its code gets executed and a value may be returned to the sender. If the contract was written in a high-level language and the message's payload properly specifies a function and parameters, the function is executed with the supplied parameters. Information about the current message and the originating transaction can be accessed from within the contract code [18], [19]. A full listing of the information that is available is given in appendix A.

Contract creation New contracts can be created both by users and existing contracts [18]. The message is then sent to a special null address, with the payload containing the byte code used to instantiate the new contract [19].

2.1.4 Execution costs

All transactions are executed by nodes participating in the network known as *verifiers*. Since the EVM is Turing-complete, contracts may enter infinite loops, which are impossible to detect in the general case due to the halting problem. This would open up for attacks in which execution never finishes, rendering verifiers unable to process subsequent transactions [18]. To prevent this, users have to pay for each execution step they trigger, at a fixed price [19]. They do this using *gas*, a resource which is bought with ether [18]. The details are not important, just note that ether in the form of gas is necessary to execute contracts and that the amount of gas supplied in the message limits the length of the computations it triggers.

```
1  contract RPS {
2    uint nPlayers;
3    mapping (address => uint) moves;
4
5    function addPlayer(uint move) returns (boolean success) {
6      if (nPlayers < 2 && msg.value >= 10) {
7        moves[msg.sender] = move;
8        nPlayers++;
9        return true;
10     else {
11       return false;
12     }
13   }
14 }
```

Figure 2.2: A contract that erroneously encodes a simple rock-paper-scissors game played for ether. Displayed is the function for players to join the game. Variables are automatically initialized to 0. The fee for joining is 10 ether as seen in the second condition on line 6, and the moves are encoded as 0, 1 or 2. Adopted from [14].

2.2 Common errors in smart contracts

There are at least four classes of errors that are common for developers who are trying to develop smart contracts on Ethereum [14]. We have identified three of these as feasible to mitigate on the language level, presented below.

2.2.1 Unexpected states

Because executing contracts is expensive, smart contracts are mainly used to encode parts of applications which have something to gain from being distributed or publicly verifiable and enforceable, most commonly parts of backend business logic. Since anyone with a key pair can send messages to any other accounts, developers have to consider unexpected input when developing contracts. This combination might be unfamiliar to software developers not used to web services and REST interfaces, as business logic in traditional systems usually resides in software components which are never publicly accessed from the outside. If the developer is not careful, unexpected input can lead to unforeseen and faulty contract states.

Consider for example fig. 2.2, which encodes a simple rock-paper-scissors game where each player bets 10 ether[4]. The two players join using the function on lines 5-13, but after they have entered the game, it is still possible for a third player to call `addPlayer`. This will fail because of the first condition on line 6: there are already two players in the game. However, any ether that the third player included in the call is not returned in the else-branch, and will be locked in the contract with no way of retrieving it. This is clearly not the developer's intention, despite the game logic being encoded correctly. Solving this is simple, as shown on line 5 of fig. 2.3. Still, discovering this type of errors can be hard, especially in more complex contracts.

[4]In fact, the players don't bet 10 ether, but 10 *wei*, the smallest subdenomination of ether. All in-code ether values are denoted in wei to avoid dealing with floating point numbers. This distinction is not important for the discussion of our work, so we ignore it for the sake of simplicity.

```
1  function addPlayer(uint move) returns (boolean success) {
2    if (nPlayers < 2 && msg.value >= 10) {
3      // ...
4    else {
5      throw; //exits and returns all ether to sender.
6    }
7  }
```

Figure 2.3: Line 5 ensures that any included ether is returned if the game is full. Adopted from [14].

2.2.2 Failure to use cryptography

Refer again to fig. 2.2. Because everything on Ethereum is completely public, it is possible for the second player to see the choice of the first player before they make their move. The way to solve this is to use cryptographic *commitments*, as shown in fig. 2.4. A commitment allows a user to commit to a secret value which can only be revealed at their discretion. The revealed value is guaranteed to be the value the user committed to, under standard cryptographic assumptions [14]. This is an adequate solution, but forcing developers to implement the same cryptographic schemes over and over again is both tedious and error-prone.

2.2.3 Full call stack

The call stack of the EVM is limited to a size of 1024. If a contract attempts to call a function when the call stack is full, it will raise an exception [19]. This includes all messages to other accounts, including sending them ether. At the moment, however, the EVM handles exceptions by simply returning 0 whenever a call fails. Additionally, exceptions do not abort execution of the calling contract [19]. Contract developers are expected to verify return values in the calling function.

This opens up the door for attacks. Consider `buyLicense` in fig. 2.5. If it is executed with a full stack, `owner.send()` will fail, leaving the owner without its ether. Additionally, the return value is never checked to see if an exception occurred, so the rights will be granted to the caller without compensation to the owner.

```
1  contract RPS {
2    mapping (address => hash256) commits;
3    mapping (address => uint)    moves;
4
5    function addPlayer(hash256 commit) returns (boolean success) {
6      // ...
7    }
8
9    function open(uint nonce, uint move) {
10     if (sha3(move, nonce, msg.sender) == commits[msg.sender]) {
11       // ...
12       moves[msg.sender] = move;
13     }
14   }
15 }
```

Figure 2.4: An example demonstrating how cryptographic commitments can be used to obscure secret data. As seen on lines 6 and 11, a commitment is a hash of the secret data and a secret nonce chosen by the sender. We also include the sender's address to gain an extra level of security if the nonce should be compromised. Adopted from [14].

```
contract LicenseManager {
  address owner;
  address[] licensees;
  uint nLicensees;

  function buyLicense() {
    if (msg.value >= 10) {
      owner.send(msg.value);
      licensees[nLicensees++] = msg.sender;
    }
  }
}
```

Figure 2.5: A contract handling licences to use some resource. The resource is owned by `owner`, with the contract granting access to anyone who pays at least 10 ether and forwards it to the owner.

This is easily solved by calling a dummy function `stackOk` in the beginning of all externally accessible functions. It takes no arguments and returns `1` by definition; if it returns `0`, the stack is full and the calling function should abort. We don't think contract developers should have to keep this in mind.

2.3 Type systems detect errors

Simple type systems only distinguish between primitive types such as integers, strings and booleans. Still, they help detect errors, since mistakenly interpreting a value of one type as another almost certainly leads to undesired results [25]. Many modern programming languages also support types which are parametrized on other types, so called generic or polymorphic types. A simple example is `List a`, the type of lists whose elements are of type `a`. This allows developers to write generic functions over lists, disregarding their content [25]. For example, `head : List a -> a` is a function which returns the first element of any non-empty list and causes a runtime error for empty lists. We adopt the jargon of the functional paradigm and will refer to these types as polymorphic rather than generic.

2.3.1 Dependent types

Some type systems also support *dependent types*, parametrized by values [26]. The canonical example of a dependent type is the type `Vect n a`, which are vectors of length `n` whose elements are of type `a`. Note that `n` is a natural number, while `a` is a type. This has the effect that `Vect 5 Int` and `Vect 4 Int` are distinct types which are incompatible with each other; a function operating exclusively on one of them would not accept the other. To give a further intuition of how dependent types are used, consider again the function for retrieving the first element, but this time on vectors: `head : Vect (S n) a -> a`. Here, `S` constructs the successor of a natural number, while `n` is a variable denoting any natural number. Thus, the expression `S n` denotes "a natural number that is the successor of any natural number", i.e. any natural number except 0, which by definition is *not* the successor of any natural number. This has the interesting result that `head` is only defined on non-empty vectors and can never fail at runtime.

There are two reasons why we are interested in dependent types. First, they can detect yet another class of errors at the language level. As we just saw, it is possible to give much more fine-grained knowledge to the type checker, making invalid or unexpected input simply not pass type-checking. Second, there exist an isomorphism between logical propositions and their proofs on one hand, and types and programs on the other. That is, any type signature can be seen as a logical proposition, which is proven by a program that implements the signature, and vice versa [27]. This is true for *all* types, but the ability to include values in types, as is provided by dependent types, allows us to state much more interesting propositions.

As an example, consider fig. 2.6, displaying a type `a = b` which can only be instantiated if `a` and `b` are the same value. A value of type `a = b` is a *proof* that $a = b$, which has interesting consequences. Consider the function `f` on line 4. The first two arguments are integers that are bound to the names `a` and `b`, which are then used as parameters to the type of the third argument. Just by looking at the type signature, we can be absolutely certain that `f` will *never* be executed if the first two arguments are not equal, because then it would be *impossible* to pass the third argument.

```
1 data (=) : a -> a -> Type where
2    Refl : x = x
3
4 f : (a : Int) -> (b : Int) -> a = b -> c
```

Figure 2.6: A type parametrized on two values of type `a`. The sole inhabitor of the type is `Refl`, which can only be constructed if the two parameters are in fact the same value `x`.

In chapter 3, we will demonstrate in detail how polymorphic and dependent types can play a central role in solving or detecting the first two classes of common errors described above. Before that, we will take a closer look at Idris, the dependently typed functional language which we use to demonstrate our proposed solutions.

2.4 The Idris programming language

Idris is a dependently typed and purely functional programming language, still under development [28]. As can be seen in fig. 2.7, its syntax is very similar to that of Haskell. Specifically: functions are declared as equations; function application is parenthesis-free; functions are curried; algebraic data types are supported, which can be pattern matched. It is also statically typed, so type errors are caught at compile time rather than at runtime. While Haskell employs a lazy evaluation strategy, Idris uses eager evaluation [28]. A minor syntactic difference is that type signatures are specified by a single colon (`:`), while construction of lists and vectors use double colons (`::`).

```
1 vadd : Vect n Int -> Vect n
    Int
2 vadd []      []      = []
3 vadd (x::xs) (y::ys) = x+y :: vadd xs ys
```

Figure 2.7: An Idris function implementing vector addition. Because the input vectors are guaranteed to be of the same length, the base case on line 2 is sufficient.

```
V8 : Type -> Type
V8 t = Vect 8 t
```

Figure 2.8: A function that acts as a polymorphic type synonym for vectors of length 8.

One important thing to note is that types are first class citizens of the language and can be treated just as values and functions can in Haskell [28]. A small example demonstrating this is the function `V8` displayed in fig. 2.8, which is a function computing a type. The expression `V8 Int` will evaluate to `Vect 8 Int`. Because types are first class citizens, type synonyms can be implemented as functions in this way and are not a language primitive. As shown in the type signature of `V8`, types have the type `Type` and can both be passed as arguments and returned.

2.4.1 Side effects

Because purely functional languages exclusively deal with functions in the mathematical sense—i.e. objects which *only* map input to output, also known as *pure* functions—a major concern is how to represent side effects. These include actions such as mutating a global state or accessing the system's I/O capabilities, which is not possible for a pure function to do, but are instead described by *effectful* functions. When the distinction is necessary, we will refer to effectful functions as *operations*, while pure functions will be referred to simply as functions. Operations are necessary components of many programs, including most smart contracts on Ethereum, which is why their representation in the language is of great interest to us. This section will explain how Idris uses and represents operations, with a focus on the features that we make use of.

Syntax for operations

Just as Haskell, Idris offers `do`-notation to sequence operations together [28]. This resembles the style of imperative languages, which is useful for functions that are "very" effectful. However, a large class of operations only need effectful values once. As shown in fig. 2.9, in these cases `do`-notation leads to verbose code, which can obscure the functionality. Of course, one could resort to the more terse applicative-style programming as in fig. 2.10, but not all functional programmers are used to this style and it is arguably quite hard to read.

```
addM = do
  a' <- a
  b' <- b
  return (a+b)
```

Figure 2.9: Using `do`-notation to access effectful values.

```
addM = (+) <$> a <*> b
```

Figure 2.10: Using applicative style to access effectful values.

```
addM = return (!a + !b)
```

Figure 2.11: Using `!`-notation to access effectful values.

Idris solves this with a bit of clever syntactic sugar called `!`-notation. The expression `!a` implicitly binds the value of `a` to the expression as a whole. For example, the expression `f !a` is desugared to: `a >>= \a' => f a'` [5] [29]. Thus, the previous example could be written as in fig. 2.11, clearly showing the intended functionality in a concise way while also maintaining purity.

[5]Idris uses a thick arrow `=>` for lambda bindings, in contrast to the thin `->` in Haskell.

```
get :       Eff s  [STATE s]
put : s -> Eff ()  [STATE s]
```

Figure 2.12: The types of the fundamental operations for stateful computations. `get` retrieves the state, while `put` updates it [29].

```
readPush : Eff ()
      [STATE (Vect n a),     STDIO]
      [STATE (Vect (S n) a), STDIO]
```

Figure 2.13: The type of an operation that reads a value from the standard input and adds it to a vector state [29]. Because vectors of different lengths have different types, the effect has to change as well.

Handling of effects

Idris represents operations using an algebraic representation called *effects* [29], initially pioneered by the Eff language [30]. Effects define which side effects can be used, while exactly how they are achieved is a separate matter and is defined by *handlers*[6]. Each effect can have one or more handlers, each one interpreting the effect in a certain *computational context* [29]. A computational context can be viewed as a place where values can exist and computation can occur, but these may result in other things as well. For example: the `Maybe` context represents computations that may fail and the `IO` context represents computations that may perform I/O. A value of type `Maybe Int` is not simply an integer but may also represent a failed computation and a value of type `IO Int` is an integer whose value is dependent on the I/O that was performed while computing it.

As mentioned, effects are interpreted in computational contexts. For example, an effect to read and write files could be interpreted in the context of an I/O-library if the files are stored on the same computer. It could also be interpreted in the context of a library handling network communication if the files are stored on other computers. Both of those contexts are impure, but effects themselves are always pure. Effects and handlers separate the description of side effects from the execution of them, allowing for easier reasoning about their results and more modular code.

Not all effects are handled in impure contexts. The only requirement is that the type of the context is `Type -> Type`, so handling effects in pure contexts like `Maybe` is possible [29].

Dependent algebraic effects

As a concrete example of how operations are described, consider fig. 2.12. It shows the type signatures of the two fundamental operations of the effect that models stateful computations, analogous to the state monad in Haskell. That the functions are operations can be seen from the return types being on the form `Eff a effs`. The first argument to `Eff` specifies the type of the return value; `get` returns the current state and `put` doesn't return anything. The second argument is a list of the effects that the operation has access to. Here, the only available effect is `STATE s`, allowing access to a shared mutable state of type `s` [29].

It is also possible to let an operation change the available effects. As seen in fig. 2.13, this is done on the form `Eff a ineffs outeffs`, made possible because `Eff` has a number of overloaded alternatives distinguished by their types. As before,

[6]Advanced readers might recognize this as being analogous to using free monads to describe effectful computations, although as we will see shortly, Idris effects offers greater functionality.

the second argument is a list of the effects that are accessible when the operation is called, while the third argument is the list of the effects that will be accessible to the next operation [29]. That the operation actually complies to the specification given by the type signature is verified by the type checker. The example also shows how multiple effects can be used by the same operation, with both `STATE` and `STDIO` being available.

In the previous example the output effects were static, but they can also depend on the result of the operation, as can be seen in fig. 2.14. This is the same operation as in the previous example, but we have extended the type signature to take into account that the reading operation may fail. Result-dependent effectful operations have types on the form `Eff a ineffs reseffs`, where the third argument is a function with the type `a -> List EFFECT`. The operation's return value will be passed to `reseffs`, which determines the output effects [29]. This allows for very powerful specifications, as we will see in the next chapter.

```
readPush : Eff Bool
                [STATE (Vect n a), STDIO]
                (\ok => if ok then [STATE (Vect (S n) a), STDIO]
                                   [STATE (Vect n a),     STDIO])
```

Figure 2.14: The type of an operation that reads a value from the standard input and adds it to a vector state, taking into account that the reading may fail [29]. Because vectors of different lengths have different types, the effect should only change if the reading was successful.

3
Smart contracts in Idris

This chapter will show how polymorphic and dependent types can be used to allow safer development of smart contracts for Ethereum. We will demonstrate how the common errors of unexpected states and failure to use cryptography described in section 2.2 can be detected or solved more easily using an Idris library that we have defined. This library is used by developers to interact with the Ethereum system, and can help ensure that certain properties are satisfied. It will be presented gradually, with examples showing the intended usage.

3.1 Avoiding unexpected states

This section will introduce two effects from the Idris library we have defined. They are used to handle ether and to access environment variables. We will use them to incrementally increase the complexity of a running example contract that acts as a bank. This will show how the library we have defined can be used to ensure that the correct amount of ether is transferred and kept in all execution paths, ultimately avoiding the unexpected states noted in section 2.2.1.

3.1.1 Handling ether

Since sent and received ether are neither function arguments in the strict sense nor part of the function output, we manage ether using an effect. The `ETH` effect shown in fig. 3.1 offers a few operations for managing ether, namely `value`, `balance`, `contractBalance`, `keep` and `send` ("ETH" is the currency code for ether, hence the name of our effect). Their functionalities are explained in their respective comments. What is of particular interest is the parameters of the effect itself as seen on lines 1-5. These specify the amount of ether included in the call, the contract's balance, the amount of ether transferred during the current execution and the amount of ether explicitly kept during the current execution. The first two are static during a particular execution, but their values can be used to specify how the other two should change[1]. The other two change only when the operations `keep` and `send` are used, as can be seen in the type signatures of these operations. The amount of ether transferred is increased by `send`, while `keep` increases the amount kept.

[1]As the reader might notice and as we will see shortly, this leads to a slightly redundant syntax where the first two parameters are repeated. This is due to a design limitation in Idris effects. Our library includes an alternative syntax for less redundant effect specifications, presented later.

```
ETH : (value : Nat)    -- Incoming ether (from call)
   -> (balance : Nat) -- Ingoing contract balance (since call)
   -> (trans : Nat)   -- Ether sent in outgoing transactions
   -> (kept : Nat)    -- Amount of "value" explicitly kept
   -> EFFECT

-- Incoming ether (from call).
value : Eff Nat [ETH value balance trans kept]

-- Balance of a given contract.
balance : Address -> Eff Nat [ETH value balance trans kept]

-- Ingoing contract balance for this contract (since call)
contractBalance : Eff Nat [ETH value balance trans kept]

-- Explicitly keep amount of incoming ether to contract balance.
-- Note that all incoming ether is already included in contractBalance
-- and that this compiles to a no-op. This function is used by the
-- type checker to ensure that the contract complies to the type
-- specification.
keep : (amount : Nat) -> Eff ()
                         [ETH value balance trans kept]
                         [ETH value balance trans (kept+amount)]

-- Send amount of ether to given recipient.
send : (amount : Nat) -> (recipient : Address) -> Eff ()
       [ETH value balance trans          kept]
       [ETH value balance (trans+amount) kept]
```

Figure 3.1: The ether effect and its operations.

```
1 deposit : Eff ()
            [ETH v b 0 0]
            [ETH v b 0 v]
2 deposit {v} = keep v
```

Figure 3.2: A simple ether storing contract.

We will now have a look at a trivial ether handling contract, displayed in fig. 3.2. It keeps any ether it receives and does nothing else. As can be seen on line 2, its only operation `deposit` takes one parameter `v`, representing the amount of ether received in the call. The braces signify that `v` is an *implicit* argument which is inferred from the context; it is part of the transaction but not of the function arguments.

Now, let's have a look at the type signature of `deposit` on line 1. We will from here on out use single-letter names for effect parameters, as they will always have the same meaning and occur in the same order. The only available effect in this example is one that handles ether, with the input effect `ETH v b 0 0` signifying an input of `v` ether and a contract balance of `b` ether. Since this is supposed to be the entry-point, no ether should have been kept by or transferred from the contract during the call yet, hence the last two parameters are set to `0`. It is the inclusion of `v` in the type signature that allows us to treat it as an implicit argument in the implementation on line 2, a standard feature of Idris [28]. Moving on to the output effect, `ETH v b 0 v`. As explained previously, `v` and `b` have to have the same values as in the input effect. The third parameter means that `0` ether has been transferred from the contract, and the last one that `v` ether has been kept by to the contract. An operation implementing this type has to follow these requirements in order to type check, hence the developer is required to explicitly keep the received ether. We can already here see how the combination of dependent types and algebraic effects can ensure that the implementation works as intended.

```
ENV : (sender : Address) -- Sender of transaction (current call)
   -> (origin : Address) -- Origin of transaction (full call chain)
   -> EFFECT

-- Address of this contract
self : Eff Address [ENV sender origin]

-- Sender of transaction (current call)
sender : Eff Address [ENV sender origin]

-- Origin of transaction (full call chain)
origin : Eff Address [ENV sender origin]

-- Gas remaining for this call
remainingGas : Eff Nat [ENV sender origin]

-- Timestamp of the current transaction
timeStamp : Eff Nat [ENV sender origin]
```

Figure 3.3: The environment effect and some of its operations.

3.1.2 The execution environment

The previous example is rather pointless by itself, since any ether deposited is effectively stuck in a black hole. Let's say that we wanted to add an operation to allow a hard-coded address to withdraw an arbitrary amount of ether. For this, we need to know the sender of the current call. We offer the effect `ENV` (short for environment) to allow the developer to access details of the execution environment such as the sender of a call, the contract's address, the current time and so on. It is partially displayed in fig. 3.3, with some clarifying comments. As can be seen, it is parametrized on the sender of the current call and the origin of the transaction. Just like the value and balance parameters to `ETH`, both of these are static throughout an execution; they are only included to allow certain properties to be enforced by the type checker, as we will see next.

The usage of `ENV` is demonstrated in fig. 3.4, where we use it to implement the withdrawal operation discussed above. Let's dissect the type signature, seen on line 4. The argument `amount : Nat` is a natural number indicating the requested amount to withdraw. This is a regular argument that the user calls the contract with. The second, implicit, argument is an automatically constructed proof `p` that the requested withdrawal amount does not exceed the contract's balance (`LTE` stands for "less than or equal" and is written in prefix form). If such a proof is not supplied and cannot be constructed, the arguments are not type-correct, hence the operation will not execute and any ether should be returned to the sender.

Moving on to the effects available to `withdraw`, `ETH` and `ENV`. Of particular interest is the effect `ENV Owner o`. The first parameter, `Owner`, has been previously defined on lines 1-2 and is thus required to be this particular value. In this way,

```
1 Owner : Address
2 Owner = 0x00cf7667b8dd4ece1728ef7809bc844a1356aadf
3
4 withdraw : (amount : Nat) -> {auto p : LTE amount b} -> Eff ()
             [ETH 0 b 0      0, ENV Owner o]
             [ETH 0 b amount 0, ENV Owner o]
5 withdraw amount = send amount Owner
```

Figure 3.4: A withdrawal operation that checks sender address and contract balance.

we specify that only this particular account is allowed to call this function. The second parameter is not important in this particular implementation. Furthermore, the input effect `ETH 0 b 0 0` and output effect `ETH 0 b amount 0` ensure that no ether is attached to the call and that the requested amount is sent. To summarize, we have now enforced the following properties using only the type signature:

- No ether is attached to the call.
- The requested amount does not exceed the balance.
- The requested amount has been sent when the operation is finished.
- Only a specified address is allowed to withdraw.

Given this, the implementation is incredibly simple. Since these conditions are enforced in the type, we can safely send the requested ether to the, already known, sender. Also note the return type: `()`. Similar operations would usually return a boolean value indicating success. Here, this is not necessary since the function always succeeds if it can be called. This is a fairly trivial example where the need to check for edge cases in the implementation is removed, as problematized in section 2.2.1. The same methods are applicable for more complex operations as well, as will be seen in section 3.4.

Using dependent effects elegantly

To expand on the bank example, let's see how we can accommodate for more than one owner. This is something we cannot check in the signature as we did with the case of a single owner; the verification has to be performed in the implementation. As a consequence, the return type is changed to `Bool` since the function can now fail. The implementation of this is given in fig. 3.5, with line 4 showing how the result of the operation is used to determine the resulting effects. Using dependent effects to differentiate on the output value, we can still let the type system ensure that ether only gets sent out if the function succeeds.

```
1 Owners : List Address
2 Owners = [0x00cf7667b8dd4ece1728ef7809bc844a1356aadf
           ,0x004a7617b84d4ece1728ef7809bc844356a897ba
           ]
3
4 withdraw : (a : Nat) -> {auto p: LTE a b} -> Eff Bool
             [ETH 0 b 0 0, ENV s o]
             (\success => if success
                             then [ETH 0 b a 0, ENV s o]
                             else [ETH 0 b 0 0, ENV s o])
5 withdraw a {s} = if s `elem` Owners
6                    then do
7                      send a s
8                      pureM True
9                    else pureM False
```

Figure 3.5: A withdrawal operation that allows for different owners.

```
namespace Field
  data Field a = MkField Nat

namespace MapField
  data Field a b = MkField Nat

STORE : EFFECT

namespace Field
  read   : Field a -> Eff a [STORE]
  write  : Field a -> a -> Eff () [STORE]
  update : Field a -> (a -> a) -> Eff () [STORE]

namespace MapField
  read   : Field a b -> a -> Eff b [STORE]
  write  : Field a b -> a -> b -> Eff () [STORE]
  update : Field a b -> a -> (b -> b) -> Eff () [STORE]
```

Figure 3.6: The store effect with its operations. Note that this effect has no parameters and therefore doesn't have any use for enforcing invariants in type signatures.

3.2 Using the data store

Handling ether and retrieving environment variables are not the only side effects a contract can have. As mentioned, they also have a persistent memory, which is accessed using the `STORE` effect, shown in fig. 3.6. The data store is accessed through *fields*: simple memory references, parametrized by the type of the referenced value. There are simple fields, referencing primitive values, and map fields, referencing mappings from one primitive type to another. The values referenced by fields are accessed using the functions `read`, `write` and `update`.

Extending the bank example in fig. 3.7, we use `STORE` to extend the functions from the previous examples to accommodate for any account to use this contract as a bank. To achieve this, we map addresses to their respective balances using the field `balances`, defined on lines 1-2 and accessed on lines 6, 11 and 13.

```
 1 balances : Field Address Int
 2 balances = MkField 0
 3
 4 deposit : Eff ()
              [STORE, ETH v b 0 0, ENV s o]
              [STORE, ETH v b 0 v, ENV s o]
 5 deposit {s} {v} = do
 6     update balances s (+v)
 7     keep v
 8
 9 withdraw : (a : Nat) -> Eff Bool
              [STORE, ETH 0 b 0 0, ENV s o]
              (\success => if success
                              then [STORE, ETH 0 b a 0, ENV s o]
                              else [STORE, ETH 0 b 0 0, ENV s o])
10 withdraw a {s} =
11     if !(read balances s) >= a
12         then do
13             update balances s (\b => b - a)
14             send a s
15             pureM True
16         else pureM False
```

Figure 3.7: Balance is now individualized for each user calling the contract.

3.3 Annotating for cryptography

We noted in section 2.2.2 how cryptographic commitments can be used to handle secret information, but argued that developers should not have to implement these. Building on standard practices in programming language research, our proposed solution is to annotate the types of secret values using a polymorphic type `Commit : Type -> Type` [31], [32]. An illustration of how this could be done is shown in fig. 3.8

As mentioned, a commitment is a hash of the secret value, a nonce and possibly other information. In general, developers will want to save any commitments received by a contract to the data store. The sender can then reveal the secret value at a later time by providing the values that were used to create the commitment. Once revealed, the now public value can be accessed using the library function `open : Commit a -> Maybe a`, as on line 8. If the commitment has not been revealed previous to calling `open`, it will return `Nothing`.

```
1 secret : Field Int
2 secret = MkField 0
3
4 submitSecret : Commit Int -> Eff () [STORE]
5 submitSecret newSecret = write secret newSecret
6
7 getSecret : Eff Int [STORE]
8 getSecret = fromMaybe 0 (open !(read secret))
```

Figure 3.8: A simple contract for saving a secret value. After it has been revealed (not shown here), it can be accessed by anyone.

3.4 Reimplementation of rock-paper-scissors

In section 2.2.1, a rock-paper-scissors game from [14] is mentioned. Below is a full version of the game implemented in Idris using our standard library. It illustrates how two of the identified problems can be prevented by correctly specified type signatures. Specifically, note the type of `joinGame` on line 25—requiring the implementation to explicitly keep and send ether on lines 33, 34 and 37—and the use of the `Commit`, which is followed by the opening of the players' moves on line 42. Also note that invalid moves result in a draw. We leave it as an exercise for the interested reader to enforce this using the type system.

```
import Effects
import Ethereum

playerCount : Field Int
playerCount = MkField 0

players : Field Int Address
players = MkField 1

moves : Field Int (Commit Int)
moves = MkField 2

winner : Int -> Int -> Int
winner 2 1 = 0 -- Scissors beats paper.
winner 1 2 = 1 -- Scissors beats paper.
winner 0 2 = 0 -- Rock beats scissors.
winner 2 0 = 1 -- Rock beats scissors.
winner 1 0 = 0 -- Paper beats rock.
winner 0 1 = 1 -- Paper beats rock.
winner _ _ = 2 -- Draw

init : Eff () [STORE]
init = write playerCount 0

joinGame : {auto p : LTE 10 v} -> Commit Int -> Eff Bool
           [STORE, ETH v b 0 0, ENV s o]
           (\succ => if succ
                         then [STORE, ETH v b (v-10) 10, ENV s o]
                         else [STORE, ETH v b v      0 , ENV s o])
joinGame {v} {s} move = do
  pc <- read playerCount
  if pc < 2
   then do
      write players pc s
      write moves pc move
      write playerCount (pc+1)
      keep 10
      send (v-10) s
      pureM True
    else do              -- If the game is full, return ether
      send v s
      pureM False

finalize : {auto p: LTE 20 b} -> Eff Int
           [STORE, ETH 0 b 0 0]
           (\winner => if winner == 0
                           then [STORE, ETH 0 b 0  0]
                           else [STORE, ETH 0 b 20 0])
finalize = if !(read playerCount) == 2
    then case (open !(read moves 0), open !(read moves 1)) of
           (Just m0, Just m1) => do
             write playerCount 0
             case winner m0 m1 of
               0 => do send 20 !(read players 0)
                       pureM 1
               1 => do send 20 !(read players 1)
                       pureM 2
               _ => do send 10 !(read players 0)
                       send 10 !(read players 1)
                       pureM 3
         _ => pureM 0
    else pureM 0
```

Figure 3.9: Rock-paper-scissors implemented in Idris using our Ethereum library.

4 Implementation

To show that the methods we have proposed are realistic, we have extended the Idris compiler with a code generator targeting the EVM, as well as implemented a library to allow necessary system interaction[1]. This chapter presents these and the choices made when implementing them, along with some necessary details on the technologies we are relying on. It is worth reiterating that the implementation is not supposed to be readily usable in production, but merely serves as a proof-of-concept.

This chapter will start with a discussion of the target and source languages used by our code generator, followed by an explanation of its implementation. Then, the components of our library will be presented and explained. Finally, a few details on how to use our implementation are explained.

4.1 Target language

When implementing a new language for the EVM, there are currently four possible target languages for the code generator: straight to assembly instructions/bytecode or through one of the three existing higher-level languages. Two of the higher-level languages support assembly instructions—except jumps—in addition to their higher-level constructs, essentially offering a superset of the functionality to the first option [19]. Furthermore, the EVM will most likely be redesigned in the near future, potentially including breaking changes to the assembly language. The higher-level languages will continue working as usual [33].

This caused us to discard the going directly to assembly instructions. Hence we are left with the following alternatives, ranging from lower to higher levels of abstraction: *LLL* (short for Low-Level LISP), intended as a low-level language for compilation [19]; *Serpent*, a language with Python-inspired syntax that is more low level and minimal than its source of inspiration, but not as much so as LLL [18]; *Solidity*, the most widely used, maintained and feature-rich language, with syntax similar to JavaScript [34]. The differences in language features relevant to us are shown in fig. 4.1.

LLL has the advantages of lower runtime overhead and smaller binary output than code written in Serpent and Solidity. Its simple syntax also makes it an appropriate bridge from the Idris compiler's intermediate representation. Finally, the existing LLL compiler offers some optimizations as part of compilation. It is quite clear to us that a production grade implementation should target LLL or byte code.

[1]These can be found at `https://github.com/vindaloo-thesis/idris-se`. Example contracts can be found at `https://github.com/vindaloo-thesis/examples`.

	Data structures	Contract functions	Macros	Opcodes	Targets
LLL	No	No	Yes	Yes	Bytecode
Serpent	Yes	Yes	Yes	Yes	LLL
Solidity	Yes	Yes	No	No	Bytecode

Figure 4.1: The three mainstream languages for smart contract development and some of their differences. The "Opcodes" column refers to the possibility of directly invoking assembly instructions, with the exception of jumps.

However, both of these alternatives come with three additional concerns, due to their low-level nature. First, the only way to access the persistent memory is through direct reading and writing to addresses. Second, there is no support for complex data types such as arrays and lists. Allocating and aligning memory positions efficiently is non-trivial, especially for complex data types, and certainly not the focus of our work. Third, there is no concept of functions, so all programs have a single entry point, requiring all user-defined functions to be implemented as conditional checks [35].

On the other hand, Serpent's compiler handles all of the above while still giving access to assembly instructions and is easier to use. It is possible to declare persistent variables, but it also has direct access to the persistent memory if needed, offering some degree of flexibility. Additionally, since Serpent compiles to LLL, a Serpent code generator should provide a reasonable reference implementation if one were to implement a code generator targeting LLL at a later time. The disadvantages are mainly a relatively minor increase in runtime overhead and code size [36]. As shown in fig. 4.1, Solidity does not provide all of these advantages.

We opted for Serpent as our target language. It was deemed a good middle-ground which will support further development, while still allowing us to avoid spending disproportionate amounts of time on implementation issues that are out of the scope for our research questions.

4.2 Source language

It is not only the target language that is important however, but also the source language. One of the selling points of Idris is that its compiler has been designed from the start to support new compiler backends. A new backend can choose which analysis, optimization and transformation steps should be performed, and then specify its own code generator. After parsing and type checking, the Idris compiler takes the code through up to six intermediate representations; code generators are free to use any of these as their input [37]. The full compiler pipeline is as follows:

Parsing and type checking Transformation of the source code to an abstract syntax tree (AST) representation that is type checked and saved as an intermediary `.ibc` file.

Intermediary transformations The AST is transformed to a lower-level language to be used as a basis for code generation. This is done by going through a series of intermediate languages of decreasing complexity, detailed below.

Optimizations are applied in some of these stages. Different backends stop at different stages, so not all stages presented below are used in all situations.

TT This is the core type theory (hence TT) that is the foundation of Idris. It is a minimal core language with the same semantics as Idris, but a subset of the syntax. All types are explicit. For the readers familiar with the internals of GHC, TT can be seen as the Idris compiler's equivalent of GHC's Core.

$\mathsf{TT}_{\mathsf{case}}$ All pattern matches have been converted to `case` trees.

$\mathsf{IR}_{\mathsf{case}}$ All types, and values provably not used at runtime, have been erased. (IR is short for "intermediate representation".)

$\mathsf{IR}_{\mathsf{lift}}$ All lambdas have been lifted to the top level.

$\mathsf{IR}_{\mathsf{defunc}}$ All functions are first-order and fully applied.

$\mathsf{IR}_{\mathsf{ANF}}$ All applications are in *applicative normal form* (ANF). Here, this means that all arguments to functions, constructors and primitive operators have to be either variables or constants, and can therefore be trivially evaluated.

Code generation After transforming the program to an appropriate intermediate language, a code generator uses the AST to generate output code.

As Serpent supports neither functions as first-class objects nor higher-order functions, $\mathsf{IR}_{\mathsf{defunc}}$ and $\mathsf{IR}_{\mathsf{ANF}}$ are the most reasonable options for the source language, since they don't require the runtime system to implement closures [37]. The difference between the two is not big. $\mathsf{IR}_{\mathsf{defunc}}$ has the advantage that function arguments are not required to be trivial, so less instructions and memory are spent evaluating them in minuscule steps. Conversely, $\mathsf{IR}_{\mathsf{ANF}}$ is extremely simple since all subexpressions are either constants or have been lifted out into variables, which are easy to map onto memory locations [37]. We chose $\mathsf{IR}_{\mathsf{ANF}}$ because of its simplicity and because there was already a reference code generator using $\mathsf{IR}_{\mathsf{ANF}}$ available [37], [38], which we modified and extended to fit our purposes.

4.3 Code generator implementation

The Idris compiler is implemented in Haskell and can be used as a library by new code generators [37]. Because of this, we opted to implement our code generator in Haskell as well. As can be seen in fig. 4.2, we take an AST directly from the Idris compiler after it has transformed the code to $\mathsf{IR}_{\mathsf{ANF}}$, which is used to generate a Serpent source file. The AST is represented by a Haskell data type and our code generator is essentially a function from this type to a string.

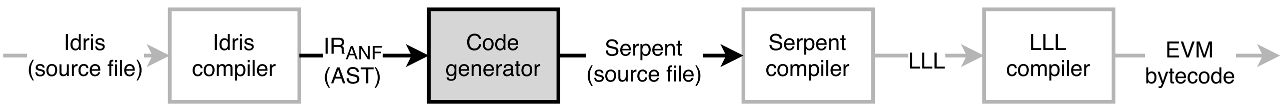

Figure 4.2: The pipeline we use for running Idris on the EVM. Our contribution is highlighted.

$$\begin{array}{rcllcll}
e & ::= & x & \text{(variable)} & | & i & \text{(constant)} \\
 & | & \mathbf{f}_{tail}\ \vec{v} & \text{(function)} & | & \mathsf{c}_i\ \vec{v} & \text{(constructor)} \\
 & | & \underline{\text{let}}\ x = e_1\ \underline{\text{in}}\ e_2 & \text{(let binding)} & | & op\ \vec{v} & \text{(primitive)} \\
 & | & \underline{\text{case}}\ e\ \underline{\text{of}}\ \vec{alt} & \text{(case block)} & | & \underline{\text{constcase}}\ e\ \underline{\text{of}}\ \vec{calt} & \text{(case block)} \\
 & | & \underline{\text{error}}\ \langle string \rangle & \text{(abort)} & | & \square & \text{(unused value)}
\end{array}$$

$$\begin{array}{rcll}
v & ::= & x & \text{(variable)} \\
 & | & i & \text{(constant)}
\end{array}$$

$$\begin{array}{rclll}
alt & ::= & \mathsf{c}\ \vec{x} & \mapsto e & \text{(constructor case)} \\
 & | & \_ & \mapsto e & \text{(default case)}
\end{array}$$

$$\begin{array}{rclll}
calt & ::= & i & \mapsto e & \text{(constant case)} \\
 & | & \_ & \mapsto e & \text{(default case)}
\end{array}$$

Figure 4.3: The grammar of $\mathsf{IR}_{\mathsf{ANF}}$, showing expressions ($e$), values ($v$) and case alternatives (alt, $calt$) [37].

The type of the AST is essentially a list of declarations, each specifying a list of bound names and an expression. Expressions are specified by the grammar displayed in fig. 4.3. Most of the derivation rules should be pretty self-explanatory, with the exception of the two different rules for case blocks. To offer more flexibility for code generators, cases on constructors have simply been separated from cases on constants at the syntax level [37]. This distinction is not necessary for our target language.

Our translation from $\mathsf{IR}_{\mathsf{ANF}}$ to Serpent is fairly straightforward: let bindings are translated to variable assignments; primitive operators are translated directly to their Serpent counterparts; both kinds of case blocks are translated to `if-elif-else` blocks. Three translations are less straight forward however: function definitions, function applications and constructor applications.

4.3.1 Functions as macros

With the exception of functions that are exported for external use, we represent functions using macros instead of functions. Function definitions are translated to macro definitions and function applications are translated to macro applications. The difference is that macros are applied at compile time, so any called functions are inlined by the compiler [36]. There are two reasons for this. First, Serpent implements calls to internal functions in the same way as calls to functions in external contracts. This means that a called function does not have access to any of the memory or context from the calling function. Non-primitive data types are represented internally as memory addresses, so Serpent does not directly support using them as arguments or return values [36]. Doing so would require encoding them as primitive values. In the case of e.g. multi-dimensional arrays, which are used extensively in the generated code, this is a non-trivial matter that could become very expensive. Additionally, such function calls add an unnecessary gas overhead. Second, using macros ensures that only functions explicitly exported by the developer can be called externally.

However, there is a major disadvantage to this approach. Recursive macros will explode infinitely and crash the Serpent compiler, severely reducing the set of computations our language implementation is able to describe. This is an unfortunate but necessary consequence of our time constraints on one hand, and our choices of source and target languages on the other. We will not discuss this further at this point, but defer it until the discussion in section 5.2.2.

4.3.2 Algebraic data types as lists

Constructor applications are translated into lists, where the first element is a *tag*, an integer representing the constructor that was used, and the subsequent elements are the constructor arguments. For example, the value `Left "foo"` would be represented as `[0, "foo"]`, while `Right "bar"` would be represented as `[1, "bar"]` and `Just (Left 3.14)` would be `[1, [0, 3.14]]`. This is possible due to the loosely typed nature of Serpent. A necessary consequence of this representation is that constructor case blocks are simply translated to conditional checks on the first element of the list, i.e. conditional checks on the value just as in constant case blocks. There are some further details of interest in our code generator implementation, but these relate to our library which will be discussed next, hence we defer this until then.

4.4 Our Ethereum library

Our library mainly consists of two components:

- A number of EVM-specific primitive operations. These are not Serpent-specific but are agnostic to the target language.
- The three effects `ETH`, `STORE` and `ENV`, used by contract developers to interact with the EVM. This is the component that is exposed to developers. We also offer a simple simulation environment to run these on a local machine for simpler testing.

This section will explain how these components are implemented and how they interact with each other and the code generator.

4.4.1 Primitive operations

Since recently, Idris supports the definition of custom primitive operations by specifying their names and type signatures as external definitions [39]. This is used when Idris is compiled to environments which support operations other than the usual arithmetic and boolean operators. We use this to define operations which map directly to operations that are specific for the EVM, such as retrieving an account's

```
-- Returns the amount of ether included in the current message
%extern prim__value : Nat

-- Returns the sender of the current message.
%extern prim__sender : Address

-- Returns the balance of the account with the supplied address.
%extern prim__balance : Address -> Nat

-- Sends the supplied amount of ether to the supplied address.
%extern prim__send : Address -> Nat -> ()

-- Reads the value of the specified field.
%extern prim__read : (f : Field) -> (InterpField f)
```

Figure 4.4: A selection of the primitive operations we have defined.

balance, sending a message to another account or retrieving the sender of the current message or transaction. Some examples of the primitive operations we have defined are shown in fig. 4.4.

All EVM-specific operations are implemented in this way, but contract developers will not use these primitives in contracts. Instead, effects have to be used in order to enforce the type checking we explained in chapter 3 and maintain purity of the language. As explained in section 4.3, these operations are simply mapped directly to their Serpent counterparts in our code generator; a code generator targeting a different language would, naturally, have to map them to the corresponding constructs in that language.

4.4.2 The Ethereum effects

As has been discussed rather extensively by now, our library comes with three effects: `ETH`, `STORE` and `ENV`. Together, these make up the interface that developers use to access Ethereum-specific functionality. We will take a closer look at `ETH`; the other two are implemented completely analogously. Its definition is displayed in fig. 4.5. As with all Idris effects, it is defined purely syntactically. Lines 1-6 define the `Ether` data type which will be used to parametrize the effect as its *resource*. This is itself a parametrized type, its only constructor `MkEth` taking four natural numbers to construct a value of the type `Ether`, where the constructor arguments are also the type parameters. It is not necessary to define a resource type for all effects, e.g. `STORE` has no parameters to keep track of.

Lines 8-17 define a data type whose constructors represent the effect's fundamental operations. Their types describe if and how they modify the parameter type. The types of the operations are specified using the overloaded convenience functions `sig`. It takes the type of the fundamental operations, the return type and, optionally, the resource type and how it changes. On line 20, the `ETH` type synonym passes both of the aforementioned types to the effect constructor `MkEff`, to construct an effect which can be used in type signatures by developers. Finally, line 22 shows how an externally usable operation is defined in terms of one of the fundamental operations.

The semantics of the effect is not mentioned in its definition, since this is dependent on the handler used to interpret it. Each of our effects has a handler which interprets its fundamental operations as primitive operations of the EVM. Because

```
data Ether : Nat -> Nat -> Nat -> Nat -> Type where
  MkEth : (value : Nat)
       -> (balance : Nat)
       -> (trans : Nat)
       -> (kept : Nat)
       -> Ether value balance trans saved

data EtherRules : Effect where
  Value   : sig EtherRules Nat
            (Ether v b t k)

  ContractBalance : sig EtherRules Nat
                    (Ether v b t k)

  Balance : Address -> sig EtherRules Nat
                       (Ether v b t k)

  Keep    : (a : Nat) -> sig EtherRules ()
                         (Ether v b t k)
                         (Ether v b t (k+a))

  Send    : (a : Nat) -> (r : Address) -> sig EtherRules ()
                                          (Ether v b t k)
                                          (Ether v b (t+a) k)

ETH : Nat -> Nat -> Nat -> Nat -> EFFECT
ETH v b t k = MkEff (Ether v b t k) EtherRules

value : Eff Nat [ETH v b t k]
value = call Value
```

Figure 4.5: The definition of our ETH effect, specifying its parameters and fundamental operations.

```
Handler EtherRules m where
  handle state@(MkEth v _ _ _) Value           k = k v state
  handle state@(MkEth _ b _ _) ContractBalance k = k b state
  handle state                 (Balance a)     k =
    k (prim__balance a) state
  handle (MkEth v b t s)       (Send a r)      k =
    k (prim__send r a) (MkEth v b (t+a) s)
  handle (MkEth v b t s)       (Keep a)        k = k () (MkEth v b t
   (s+a))
```

Figure 4.6: The generic handler of our ETH effect, implementing its fundamental operations in terms of primitive operations of the EVM.

the operations are primitive, there is no need for any constraints on the computational context, which is defined to be completely generic. However, we will see why `Maybe` is our context of choice in section 4.5.1.

The handler interpreting `ETH` on the EVM is shown in fig. 4.6. Each handler is an implementation of the `Handler` interface, parametrized by the effect's fundamental operations (`EtherRules`) and the computational context (`m`). For our purposes, Idris interfaces and their implementations can be thought of as Haskell type classes and their instances. The `Handler` interface has one function, `handle`[2]. It takes three arguments and returns a value in the context the effect is interpreted in. The first argument represents the previous value of the effect's resource. The second is the fundamental operation that should currently be interpreted. The third is a

[2]For readers familiar with monads, `handle` can be thought of as corresponding to the bind function.

continuation function[3], which takes the return value of the current operation and an updated resource, and returns the interpretation of the next operation [29].

As seen on lines 2 and 3, interpreting the operations that retrieve values that are already part of the effect's resource amounts to pattern matching on the resource. The requested operation is passed to the continuation, along with an unmodified state. Because these parameters to the effect are constant during each execution, there is no need to retrieve them using primitive operations every time; how they are determined at the start of each execution will be discussed in section 4.5.1. Lines 4 and 5 show how operations are interpreted using primitive operations, which are used as ordinary functions. Finally, line 6 shows how our `Keep` operation has no other purpose than type checking. Its interpretation does absolutely nothing except updating the resource.

```
1  EthereumEff (retVal : RetType)
2              { SENDER  = s
3              ; ORIGIN  = o
4              ; VALUE   = v
5              ; BALANCE = b
6              ; TRANS   = t
7              ; KEEP    = k
8              ; ieffs => oeffs  -- or just 'ieffs'
9              }
10
11 Eff RetType
     ([ETH v b t' k', ENV s o, STORE] ++ ieffs)
     (\retVal => [ETH v b (t'+t) (k'+k), ENV s o, STORE] ++ oeffs)
```

Figure 4.7: The `EthereumEff` syntax extension. The expression on lines 1-9 will be translated to the one on line 11 at compile time.

Alternative syntax for effect types

While discussing some of our example programs, we noted that the syntax for our effect types is redundant, since several of the parameters that are useful in the type signature are constant during each execution. As promised, fig. 4.7 presents an alternative syntax for specifying the parameters of our Ethereum-related effects. We have implemented this as a set of syntax extensions, a native feature of Idris which allows developers to specify custom syntax for certain expressions [28]. Our syntax is shown at lines 1-9 with the corresponding standard Idris syntax that this translates to on line 11. This section explains how these syntax extensions are used.

On line 1, EthereumEff signifies that the following is an operation with access to the Ethereum effects `ETH`, `ENV` and `STORE`. It will return a value of `RetType`, specified by the developer. The `ETH` and `ENV` effects will be parametrized as specified on lines 2-8. All parameters except `TRANS` and `KEEP` have to be specified by either names or values. If a parameter is given a previously unbound name, it is treated as an implicit function argument. If it is given a previously bound name or a value, the parameter has to equal this. Multiple parameters may share the same name, in which case they have to be equal. The parameters `TRANS` and `KEEP` may be given names or values in this way as well, but can also be specified by more complex expressions. These expressions may depend on any value in scope, such as names

[3]The continuation function can be thought of as the second argument of the bind operator.

bound to other parameters and the return value `retVal`. Of course, any expressions, values and bound names used to specify parameters have to be of the correct types: `Address` for the first two and `Nat` for the last four.

Either the `ENV` parameters or the `ETH` parameters may be omitted if the information they provide is not necessary in the type signature. However, their operations will still be available in the implementation as long as the return type is specified using `EthereumEff`. `ieffs` and `oeffs` are entirely optional and specify two lists of any other effects that the operation supports. The first list, `ieffs` is the input effects, while `oeffs` is the output effects. Just as with `TRANS` and `KEEP`, dependent effects may be specified simply by including `retVal` in the expression for the output effects. If the other available effects will remain constant before and after the current operation, the arrow and the list of output effects may be omitted.

As an example, fig. 4.8 shows how this syntax can be used to specify the type signature as the `withdraw` operation from fig. 3.5. Here, the parameters to `ENV` have been omitted since they are not used in the type signature, but its operations are still available in the implementation. No additional effects are required, so `ieffs` and `oeffs` have also been omitted. The contract's balance, `b`, is used in the implicit proof together with the requested amount. The return value has been named `success`, and is used in the expression determining the total amount of ether that should be sent at the end of execution.

```
withdraw : (amount : Nat) -> {auto p: LTE amount b}
         -> EthereumEff (success : Bool)
              { VALUE   = 0
              ; BALANCE = b
              ; TRANS   = if success then a else 0
              ; KEEP    = 0
              }
```

Figure 4.8: The type of the withdrawal operation from fig. 3.5, specified using the `EthereumEff` syntax extension.

4.5 Usage

When discussing our implementation so far, we have omitted two important details: how type signatures are enforced, and how the mapping between functions in Idris and functions in other contract languages is specified. This section first explains how type signatures are enforced using wrapper functions. The current implementation requires developers to implement these manually, but they should be possible to generate automatically as will be discussed in section 5.2.7. How Idris' foreign function interface is used to specify the mapping between different languages, and how this is used by developers, will also be explained.

4.5.1 Wrapper functions

We have not yet presented any way to ensure that the properties encoded in the type signatures of contract operations are fact satisfied at runtime. Our language will have to interface with other languages and we cannot assume that users and contracts only send type correct data. For example, what would happen if someone

```
1 runWithdraw : Nat -> Maybe ()
2 runWithdraw amount = case lte amount prim__selfbalance of
3   Yes p => if prim__value == 0
4               then if prim__sender == Owner
5                       then runInit
                             [MkEth 0 prim__selfbalance 0 0
                             ,MkEnv prim__self Owner prim__origin]
                             (withdraw amount {p})
6                       else Nothing
7               else Nothing
8   No _  => Nothing
```

Figure 4.9: A typical wrapper function of a constrained operation. It verifies that the effect parameters have the specified values and that any necessary proofs can be constructed.

other than `Owner` tried to execute `withdraw` in fig. 3.4, or if a required proof cannot be automatically constructed?

When we discussed these properties in section 3.1.2, we established the desired behavior: the operation should not be executed, and any ether should be returned to the sender. However, functions containing these kinds of constraints cannot be included in a contract's external interface, because the types involved have no reasonable counterparts in other languages. This is solved by defining a wrapper function for each of these functions. Figure 4.9 presents an example wrapper function, which exposes the `withdraw` operation from fig. 3.5.

As can be seen, the wrapper takes the same argument, `amount`, and tries to construct a proof that this is less than the contract's balance using the function `lte : (a : Nat) -> (b : Nat) -> Dec (LTE a b)` on line 2. The `Dec` type is used to represent decidable properties, with the constructor `Yes` taking a proof of the property and the constructor `No` taking a proof of the negation. If an affirmative proof is constructed, it is then verified that the included ether is 0 (line 3) and that the sender is equal to `Owner` (line 4). If any of these steps fail, `Nothing` is returned, which will result in the execution halting and any ether being returned to the sender. How this is achieved will be explained shortly.

Otherwise, the desired operation is run using the standard function `runInit` on line 5, which executes an operation in the current context, i.e. `Maybe`. This function takes two arguments. The first is a list of the resources to be passed to the effects. The second argument is the operation to execute, `withdraw`. Note that it is passed not only its explicit argument `amount`, but also the proof that this amount is less than the contract's balance. Since the operation is interpreted in the `Maybe` context, the return value will be wrapped in the `Just` constructor.

Because the `Maybe` type is not used by any other contract languages, it has to be removed in order for contracts written in Idris to be able to interface with other contracts. Recall from section 4.3.2 that algebraic data types are translated to Serpent lists, where the head and tail correspond to the constructor and its arguments. Our code generator adds a conditional clause to all externally accessible functions which inspects the head. If it is 0 (i.e. `Nothing`), any included ether is returned to the sender and the function throws an exception. If it is 1 (i.e. `Just`), the tail of the list is returned. In this particular case, this would simply be `()`, which is unique in that it is treated by the code generator as "no return value", even though it is a value in Idris.

4.5.2 Exporting functions

When compiling an Idris program to an executable (as opposed to a library), it will have the `main` function as its single entry point [28]. From a developer's perspective however, a contract should be able to expose one or more functions that can be called externally. This could have been an issue, because the Idris compiler will remove functions that are unreachable from `main` and try to inline any functions it uses, rendering them unreachable from the outside. Luckily, Idris has a feature called *exports*, essentially used to specify the functions that should be exported by the generated code [37]. All functions that should be accessible from outside of the contract, i.e. wrapper functions, thus have to be exported using this feature.

In order to ensure that exported functions can be handled by the code generator, the developer has to supply a *foreign function interface* (FFI). It specifies the types that are supported by the target language and how functions and types are named in it. We supply an FFI to Serpent because we use it as our target language, but the same interface should also be usable for all other languages currently implemented on the EVM because of their similarities. Because of this, we have named our interface `FFI_Eth`, in order to hide implementation details of our code generator from developers. From their view, they are exporting functions for use in Ethereum, not in Serpent.

Thanks to a number of choices in the design of the Idris compiler, defining an FFI is remarkably simple [37]. The entire definition of our interface is shown in fig. 4.10. It consists of a description of the types that can be exposed in contract functions, shown on lines 2-12. This predicate is passed to the `MkFFI` constructor on line 15, along with two types that specify how function names and types are identified in the target language. The EVM languages use simple text strings for both. This defines our FFI, which is used whenever functions are to be exported for external use. It could also be used to allow contract developers to interface with functions of other contracts, but this is not yet implemented.

```
1  -- Supported foreign types
2  data EthTypes : Type -> Type where
3    -- Primitive types
4    EthInt_io    : EthTypes Int
5    EthNat_io    : EthTypes Nat
6    EthBool_io   : EthTypes Bool
7    EthChar_io   : EthTypes Char
8    EthString_io : EthTypes String
9
10   -- Other types
11   EthUnit_io  : EthTypes ()
12   EthMaybe_io : EthTypes (Maybe a)
13
14 FFI_Eth : FFI
15 FFI_Eth = MkFFI EthTypes String String
```

Figure 4.10: Our Ethereum FFI, defining supported Ethereum types and specifying how functions and parameters are named. The types `()` and `Maybe` that lack counterparts in other contract languages are exported because they represent a lack of return value and possibility of exception, as explained in section 4.5.1.

```
runDeposit : Maybe ()

runWithdraw : Nat -> Maybe ()

my_exports : FFI_Export FFI_Eth "" []
my_exports = Fun runDeposit "deposit" $
             Fun runWithdraw "withdraw" $
             End
```

Figure 4.11: A typical export definition. It exports the wrapper `runWithdraw` from fig. 4.9 with the name `withdraw`, together with a similar wrapper for the `deposit` function.

As an example of how functions are exported, fig. 4.11 shows an export definition for the previous examples. The wrapper functions `runDeposit` and `runWithdraw` are exported and assigned the external names `deposit` and `withdraw`, respectively. All top-level values of type `FFI_Export` are handled internally by the Idris compiler, which treats any functions they reference as entry points to the program and avoids inlining them [37]. Our code generator identifies these and generates the appropriate function definitions. To specify the target environment, our FFI is used as the first parameter to the type `FFI_Export`. The other two parameters are not important here and can simply be the empty string and the empty list as in this example[4].

[4]The interested reader can learn the purposes of these parameters in [37].

5
Discussion

There are several things to be said about both our implementation and our underlying theories. We have identified both advantages and disadvantages of our theoretical approaches, but ultimately find that they fulfill their purpose of allowing safer development of smart contracts. We think that our implementation serves its purpose as a proof-of-concept, even though we didn't have the time to implement everything we suggested. However, some implementation choices were not optimal. There are many opportunities for future work in the field, ranging from fundamental research questions to extending our implementation with useful functionality.

This chapter is divided into two parts, one for the theoretical aspects of our work and one for the implementation we provide. There is no section dedicated to presenting possible future work, instead we mention these opportunities while discussing the issues they relate to.

5.1 Theory

In this section, we summarize what we believe is the main contribution of our work, namely the use of an extensive type system to model smart contracts. We explain why a system for managing side effects similar to the one found in Idris is crucial to realizing this approach. Possible ways to extend this method are briefly discussed. We also explain why the functional paradigm and its theoretical underpinnings might not be the most suitable framework for modeling smart contracts, and suggest possible alternatives to explore and evaluate.

5.1.1 Types as models

Building on examples of a simple bank-like contract, we have shown how Idris' type system coupled with our library for the Ethereum platform allows for safer development of smart contracts. It offers developers a rich framework in which to describe the intended behavior of their contracts, which catches a class of common implementation errors at compile time. In particular, we have shown how to enforce ether flow and put detailed constraints on both function input and environment variables. To see our approach applied to a larger example, section 3.4 shows a complete Idris implementation of the rock-paper-scissors game from section 2.2.1.

We believe that this way of specifying certain aspects of smart contracts' behavior in the type signatures of their functions is the main contribution of this thesis. By encoding critical properties in types, the compiler can give static guarantees that these are satisfied in *all* execution paths, without need for testing or formal verification. We have shown that some of the common errors previously identified are in fact possible to capture in this way, rendering them compilation errors instead of unexpected behavior to be discovered during usage. For programs as critical as smart contracts, this is a very important advantage.

The importance of Idris' effect system to realize this approach cannot be understated. In order for type signatures to be able to encode the relatively fine-grained behavior we want to constrain, a dependent type system is not sufficient by itself. Such a system has to be coupled with a way to give types to side effects that allows their behavior to be constrained by data, in particular the function arguments, the state of the effect itself and even the function's return value. Conversely, for our solutions to work, effects need to be combined with a dependent type system to capture all properties of interest; the combination of the two is what makes Idris crucial to our work. In order to see why Idris' way of describing side effects is crucial, we'll try to reimplement the example from fig. 3.4, but this time in Haskell. Since Haskell does not have anything resembling the effect syntax of Idris, we will assume a monadic return type.

```
 1  owner : Address
 2  owner = 0x00cf7667b8dd4ece1728ef7809bc844a1356aadf
 3
 4  withdraw : Int -> EtherT Env ()
 5  withdraw amount = do
 6    s <- sender
 7    b <- balance
 8    if s == owner && b >= amount
 9      then send amount owner
10      else do
11        v <- value
12        send v s
```

Figure 5.1: Haskell implementation of fig. 3.4.

The closest we get is shown in fig. 5.1. Since Haskell lacks dependent types, there is no way to use the types to express that only `owner` is allowed to call the contract and that the requested amount has to be smaller than the balance. Instead, these properties have to be enforced using the conditional on line 8. Furthermore, since the types don't encode the amount of ether that should be saved and kept, the sending on line 9 could be omitted without any error being raised. Finally, because the properties are enforced using a conditional instead of the type signature, an else branch has to be implemented that returns any ether to the sender, in order to maintain the same functionality.

Note that there exist proposals for solutions that would make these properties expressible in Haskell types as well [40]–[42]. However, these either require currently unimplemented extensions to Haskell's core library and compiler [41] or careful crafting of types and libraries which results in code that is more demanding of the developer and incur extra runtime overhead [42]. Finally, Idris' built in support for algebraic dependent effects makes for a much more concise and readable syntax than could be achieved even with these workarounds [28].

Furthermore, using type annotations to make the use of commitments—and potentially other common constructions—easier does not capture errors per se, but makes implementation of certain common patterns both clearer and less error-prone.

5.1.2 Extending the type system

An interesting future direction would be to analyze real-life smart contracts written in our language, in order to identify additional common errors or patterns and investigate the possibility of eliminating or supporting these using an extensive type system. For example, using type annotations similarly to how we use them to represent commitments might be useful to verify the integrity of authenticated values, perhaps also for some other basic cryptographic operations. Since commitments is so far the only commonly used pattern we have recognized where this is relevant, we focused exclusively on how to represent those at the language level.

5.1.3 Unified language for contract and client

As was noted in section 4.5.1, there is no guarantee that the input a contract receives is type correct. This seems like a waste of information, when such extensive details are encoded in the type signatures and thus potentially could be enforced automatically.

For the cases when a contract is used as a component by a larger system, this can be alleviated by the use of a unified programming language which compiles to one set of contracts and one set of applications. The type checker would then be able to verify the interactions between them. Of course, all other accounts would still be able to message the contract with incorrect input, so the data will still have to be verified contract-side at runtime. Even so, this should make development of the system as a whole both safer and smoother.

5.1.4 Handling custom tokens

Ether is not the only important token on Ethereum. In fact, the recommendation is that ether should be kept for its purpose of paying for computations through gas, while contracts can define tokens of their own that are specialized for other niches [18]. At the moment, our type system can only handle ether. It is not obvious how one would extend this to work for other tokens, since they are not handled directly by EVM-instructions and there is not yet any agreed-upon standard interface for tokens.

If tokens were standardized, it should be possible to extend the language to use this standard interface to handle other tokens. There also exists a proposal by Ethereum's main researcher to add an abstraction to the protocol that would allow any token conforming to some standard to be handled similarly to ether [43]. If this proposal is implemented, it should be easy for our type system to handle any such token.

5.1.5 Suitability of the functional paradigm

When writing smart contracts in our Idris implementation, most of the critical functionality has to exist in effectful rather than pure functions. This is because pure functions have no notion of communication, but Ethereum's execution model is based entirely on messages that are sent between accounts, which it has in common with all smart contract platforms we are aware of. Furthermore, pure functions don't directly encode program state, which is another important aspect of smart contract platforms.

As mentioned in section 1.2, one of the motivations for offering a functional smart contract language is that pure functions are very well suited for formal verification and testing. However, that most of the critical functionality have to be implemented in effectful functions largely renders this motivation irrelevant. Granted, there are static analysis methods for languages that directly support side effects as well [44], [45], so this observation does not necessarily mean that our contracts are impossible to verify. However, the advantages our language offers with respect to formal verification are less clear than we would have hoped for.

Formal verification aside, that the core language is unable to directly encode fundamental properties of the domain also suggests that the functional paradigm might not be the most suitable to use in practical development. We do think that it offers advantages over the imperative paradigm that is currently prevailing. For example, safer implementations can be achieved by explicitly isolating side effects in an enforceable way [46], [47]. To use higher-order functions to process data structures is also generally less error prone than manual looping, thanks to their lack of explicit traversal. That said, we think that further improvements can be made.

Investigating other paradigms

We believe that the problems we have discovered with the functional paradigm all hint in the direction of a language based on a process calculus. These models are designed to model concurrent and distributed systems, and view message passing as the fundamental operation of computation [48]. This directly mirrors the domain. Many of the process calculi are also suitable for formal verification and describe computations in a compositional manner [48], [49], aligning with what we observed would be beneficial in section 1.2.

Granted, most of the process calculi only describe message passing and have no direct support for a mutable state, which is also crucial in this domain. However, this can in fact be represented in a very clean way, without any need to step outside the core model. In the process calculi, a program's state is defined by the messages that the program has previously received. Again, this mirrors how states are actually updated on smart contract platforms.

Regardless of which model we believe is suitable, evaluating different models of computation—e.g. the process calculi—with respect to the domain makes for very interesting future work. So does crafting a high-level language based on a model that is deemed suitable. Since we do find our method of encoding program properties using dependent types useful, it would definitely be interesting and potentially beneficial to include dependent types in this new language. Furthermore, the process calculi have enjoyed investigations into *behavioral type systems*, allowing the types of processes to encode quite detailed properties of their behaviors [50]. These type systems should also be evaluated with respect to the domain and may be found to complement or subsume a dependent type system.

5.2 Implementation

This section discusses the advantages and shortcomings of our current implementation, as well as potential future directions. In particular, we will explain how our choice to implement side effects as primitive operations contributes significantly to both the modularity of our implementation and efficiency of the output code. We also point out consequences of—and alternatives to—our choices of source and target languages for the code generator.

5.2.1 The efficiency and modularity of primitive operations

An alternative to defining external primitive operations for side effects would have been to use the FFI to define a computational context which offered the same operations [37]. Essentially, this would amount to an implementation similar to the fragment shown in fig. 5.2. It implements an alternative I/O context, `SIO`, that communicates through our Ethereum FFI—which would now become more coupled to Serpent, hence the name change. The `balance` operation is defined as a call through this interface to a Serpent function with the name `getBalance`, with the type `Address -> SIO Int` explicitly acknowledging that it would require interaction with Serpent. Given a library of similar operations in the `SIO` context, these could be used by the handlers to interpret our effects. It could be argued that this way of implementing side effects is cleaner than defining primitive operations, since it explicitly marks the operations as impure. Our initial implementation used this approach, but we found three major drawbacks that made us revise this choice.

First, many of the operations we needed to implement were not accessed by a simple existing function in Serpent. In the case in question, there is no Serpent function named `getBalance`. The balance of a particular account is represented as a property of its address, like so: `address.balance`. It is only possible to specify function names and arguments when calling external functions through the FFI. Therefore, this approach would require some mapping from the imaginary functions to the desired language features, either by the code generator—as is done in our current implementation, but with some specific function names instead of primitive operations—or by a Serpent runtime library. What we are accessing are not merely functions in the target language, but fundamental operations of the domain.

Secondly, implementing side effects by defining handlers in terms of `SIO` operations leads to an extra level of calls compared to defining them in terms of primitive operations. This extra level is visible in the generated code, leading to effectful parts of contracts to suffer an unnecessary increase in code size. Even if optimal performance was out of our scope, we still consider reduced code size a good thing.

```
SIO : Type -> Type
SIO = IO' FFI_Eth

balance : Address -> SIO Int
balance a = foreign FFI_Eth "getBalance" (Address -> SIO Int) a
```

Figure 5.2: Fragment of a discarded implementation of side effects, using our Serpent FFI to construct an I/O context specific to Serpent.

Finally, it adds unnecessary tighter coupling to Serpent as a target language. If an implementation were to target another language, such as LLL, a new context and all of its associated operations would have to be defined. Additionally, new handlers would have to be defined for all the effects, even though the operations they should be interpreted as are completely analogous.

Given these arguments, we consider our revised choice of implementing side effects as primitive operations an important advantage of our implementation. Not only *are* they in fact primitive in this domain and more efficient, but this choice renders all parts of our library agnostic to the language targeted by the code generator, thus easily reusable for future investigations into the field.

5.2.2 Incomplete code generation

As noted in section 4.3.1, there is a major problem with our current implementation, which translates functions to Serpent macros. Since these are inlined by the Serpent preprocessor, recursive functions result in the code size exploding until the Serpent compiler crashes. This would be a big problem for any purely functional language, which need recursion to be able to execute the same instructions repeatedly, but it is actually even worse for us.

When the Idris compiler takes the program through the intermediate representations, functions that are not fully applied are deferred to an eval/apply-recursion at the end of the execution [37]. All effectful functions consisting of more than one of the effect's fundamental operations uses a partially applied function, causing recursion in the resulting code. This renders programs which include such effectful functions uncompilable. Our code generator currently omits the eval/apply section of the generated code, allowing us to run some very simple programs in which no operations are sequenced after each other.

We did not have the time to solve this, but do see some possible solutions:

- Targeting EVM assembly instead of Serpent. This would allow for jumps, enabling the implementation of internal function calls with maintained context, which eliminates the need for translating functions to macros.
- Rewrite the recursive eval/apply functions to a while loop.
- Implementing a way to encode complex types as primitive types, to be able to send them as arguments.
- Choosing a source language with higher-order capabilities. This should also result in smaller code size, but would require an alternative implementation of partial application and higher-order functions.

An attempt was also made to patch the Serpent compiler to only expand each macro to a maximum level of rewrites. While this would allow us to run a larger subset of contracts, it is just a hack that does not solve the underlying problem and results in large, repeating output code. A code generator realistic for production use would have to be rewritten from scratch, with revised choices of input language or output language.

5.2.3 Inefficient output programs

In hindsight, our choice to use $\mathsf{IR}_{\mathsf{ANF}}$ as the source language for our code generator was probably not optimal. Due to its extremely simple nature, programs become very large and there's a lot of redundant operations in the resulting code. The resulting code size is very important, in part because every instruction has a price in gas, but also because there's an upper limit on the total amount of gas that can be spent in the entire network during a given time interval[1]. Creating a new contract involves storing its byte code and paying gas for that storage [19]. Due to the gas limit, only extremely simple Idris contracts can be deployed to the live Ethereum network. More complex contracts written in Idris can currently only be run in a testing network with a higher gas limit. Additionally, the increase in code size makes the resulting code harder to understand and debug.

Choosing one of the higher-level intermediate languages that were initially disregarded in section 4.2 could potentially lead to less redundancy in the generated code. However, because of their higher-order nature this would also require implementation of an evaluation model instead of the currently used eval/apply implementation given by $\mathsf{IR}_{\mathsf{ANF}}$. This was deemed out of scope. The only remaining option would then have been to use $\mathsf{IR}_{\mathsf{defunc}}$ as source language. This is still a very simple language whose programs are larger and slightly more verbose than desired, but it doesn't have the requirement that all function arguments should be either variables or constants. In $\mathsf{IR}_{\mathsf{ANF}}$, all expressions passed to a function are first assigned to temporary local variables, which are then used as arguments. This results in many unnecessary variable allocations and assignments. Using $\mathsf{IR}_{\mathsf{defunc}}$, where this is not the case, would therefore lead to both a lower runtime overhead and more minimal code.

Even though our choice was not optimal, we also want to point out that the size of the resulting code is to a large extent due to the unoptimized output of the Idris compiler. We have been in contact with the creator and main contributor of Idris, who said that there are several obvious optimization steps that will be added to the compiler. Additionally, mature lower-level compilers like the GNU Compiler Collection takes care of these optimizations for the existing Idris backend, but the optimizers for Serpent and LLL are not (yet) as efficient. Should optimizations be implemented in any one of these three compilers that we are utilizing, our code size should be reduced as well.

5.2.4 Implementing language-level commitments

Due to time constraints, our work on language-level commitments is purely theoretical at this point. Our library includes definitions of the `Commit` type and the `open` function, shown in fig. 5.3, simply so that example contracts that use committed values can be type-checked.

```
data Commit a = MkCommit String
              | MkValue a

open : Commit a -> Maybe a
open (MkValue a) = Just a
open _           = Nothing
```

Figure 5.3: Our preliminary definitions of the `Commit` type and the `open` function.

[1]The limit is dynamically and continuously set by the participants in the network, through voting. Additionally, the lengths of the time intervals in which it applies vary probabilistically. For readers familiar with blockchain technology, this is a per-block limit.

However, there is no support for this in the code generator: functionality for revealing commitments is missing. We see no technical limitations that would make our ideas hard to implement. Commitments will be stored in the data store, until they are revealed and replaced by the committed value.

5.2.5 Preventing call stack errors

Due to time constraints, we have not mitigated the issues that can arise from a full call stack that were explained in section 2.2.3. Our solution to this would be to modify the code generator to add a dummy function to every contract, and have all other functions include an initial call to this. This should be trivial.

5.2.6 Generating the data store declaration

The fields in the data store are specified using values of the types `Field.Field` and `MapField.Field`, specifying simple fields and mappings, respectively. Each constructor takes a natural number as an argument, which is used to generate the address the data is stored at. For a map field, the address is constructed from a combination of this number and the key for each value.

As has been shown in example contracts using the `STORE` effect (figs. 3.7 and 3.8), our current implementation requires the developer to supply this argument manually for each field. Besides verbosity, this has the risk of accidentally mapping two different fields to the same memory location if a careless developer maps them to the same number. We deem this acceptable due to our implementation merely serving as a proof-of-concept. It should be fairly trivial to have a preprocessor generate these, in order to only require field declarations to specify names and types.

5.2.7 Generating wrapper functions

Due to time constraints when implementing our proof-of-concept compiler, the wrapper functions discussed in section 4.5.1 need to be implemented manually by the developer. This is not desirable, since it essentially requires the developer to specify constraints twice: once in the type signatures and once in the wrappers. Additionally, it allows for errors in the implementations of wrappers, reducing the safety of our language.

The goal is for a wrapper to be generated automatically for each exported function. Implementing a preprocessor that can generate this based on an export list seems fairly straight forward. The steps involved are as follows:

1. Identify any bound parameters in the type signature. Verify these using conditionals.

2. Identify any proofs in the type signature. Try to generate these and pass them as parameters.

3. If the exported function is effectful, run it using `runInit`, using primitive operations to specify the parameters.

5.2.8 Higher-order functions

Higher-order functions is arguably one of the most important features of high-level functional languages. Defining a standard interface to allow contracts to send functions as parameters to calls would therefore be a huge contribution to the field and our language. However, to find, test and evaluate different possible representations and determining a final interface is a huge undertaking in itself, which is why we haven't explored this further.

6 Conclusion

We have shown how an advanced type system can be used to allow for safer development of smart contracts. In particular, we use dependent effects extensively and show how they can encode very detailed properties of smart contract behavior, which reduces both the risk of errors and the need for testing. Additionally, we have implemented a proof-of-concept compiler and library to demonstrate that our theories are practically realizable. This software is not ready for production use due to the large size of the resulting code and incomplete implementation, but testing has shown that our theories are correct in principle. What is missing is a matter of implementation. In these matters we consider our thesis proven, even though there is still much work to be done.

However, it is not clear whether a *functional* smart contract language is as beneficial as we initially envisioned, as noted in section 5.1.5. Admittedly, the problems we have identified are dependent on Ethereum's execution model, namely that the system is based on accounts that send each other messages. Ethereum does have this in common with all smart contract platforms that we are aware of, but the field is still maturing and this may change. If it does, the functional paradigm should be re-evaluated; for now, we conclude that further research is needed to find suitable paradigms for smart contract languages. We suggest that the process calculi are evaluated. They seem to encode the domain well, while most of them also lend themselves to formal verification and compositional program implementations.

Bibliography

[1] T. Hvitved, "A survey of formal languages for contracts", in *Formal Languages and Analysis of Contract-Oriented Software*, 2010, pp. 29–32. Available at: `http://www.diku.dk/hjemmesider/ansatte/hvitved/publications/hvitved10flacosb.pdf`.

[2] S. Peyton-Jones, J.-M. Eber, and J. Seward, "Composing contracts: An adventure in financial engineering", English, in *FME 2001: Formal Methods for Increasing Software Productivity*, ser. Lecture Notes in Computer Science, J.-N. Oliveira and P. Zave, Eds., vol. 2021, Springer Berlin Heidelberg, 2001, pp. 435–435, ISBN: 978-3-540-41791-0. DOI: `10.1007/3-540-45251-6_24`. Available at: `http://dx.doi.org/10.1007/3-540-45251-6_24`.

[3] N. Szabo, *A Formal Language for Analyzing Contracts*, 2002. Available at: `http://szabo.best.vwh.net/contractlanguage.html` (visited on 2015-09-18).

[4] P. Bahr, J. Berthold, and M. Elsman, "Certified symbolic management of financial multi-party contracts", in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015, Vancouver, BC, Canada: ACM, 2015, pp. 315–327, ISBN: 978-1-4503-3669-7. DOI: `10.1145/2784731.2784747`. Available at: `http://doi.acm.org/10.1145/2784731.2784747`.

[5] N. Szabo, *Smart contracts*, 1994. Available at: `http://szabo.best.vwh.net/smart.contracts.html` (visited on 2016-02-08).

[6] ——, "Formalizing and securing relationships on public networks", *First Monday*, vol. 2, no. 9, 1997, ISSN: 13960466. Available at: `http://firstmonday.org/ojs/index.php/fm/article/view/548` (visited on 2016-02-08).

[7] M. Swan, *Blockchain: Blueprint for a New Economy.* O'Reilly Media, Jan. 2015, ch. 2 – 4, ISBN: 978-1-4919-2047-3.

[8] T. Swanson, *Great chain of numbers: A guide to smart contracts, smart property and trustless asset management*, 2014.

[9] F. Vogelsteller. (2015). Crowdfunding example contract in Solidity, Available at: `https://github.com/chriseth/cpp-ethereum/wiki/Crowdfunding-example-contract-in-Solidity` (visited on 2015-08-21).

[10] N. Dodson. (2015). Boardroom: A next generation decentralized governance apparatus, Available at: `http://boardroom.to/BoardRoom_WhitePaper.pdf` (visited on 2015-08-21).

[11] Forecast Foundation. (2015). Augur prediction market, Available at: `http://www.augur.net/` (visited on 2015-08-21).

[12] V. Pureswaran and P. Brody. (2015). Device democracy: Saving the future of the Internet of Things, IBM Institute for Business Value, Available at: `http://public.dhe.ibm.com/common/ssi/ecm/gb/en/gbe03620usen/GBE03620USEN.PDF` (visited on 2015-12-17).

[13] H. Diedrich, "IBM MTN project", Presentation, Ethereum Devcon1, 2015, Available at: `https://www.youtube.com/watch?v=_kTajbcAd9E` (visited on 2016-01-13).

[14] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab", 2015. Available at: `http://ia.cr/2015/460` (visited on 2015-09-25).

[15] P. Barrientos and P. Martinez Lopez, "Developing DSLs using combinators. a design pattern.", in *International Multiconference on Computer Science and Information Technology*, Oct. 2009, pp. 635–642. DOI: `10.1109/IMCSIT.2009.5352773`.

[16] B. O'Sullivan, D. Stewart, and J. Goerzen, *Real World Haskell.* O'Reilly Media, 2008, ch. 11. Available at: `http://book.realworldhaskell.org/read/testing-and-quality-assurance.html` (visited on 2016-02-08).

[17] L. Cardelli, "Type systems", in *The Computer Science and Engineering Handbook*, A. B. Tucker, Ed. CRC Press, 2004.

[18] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform", White Paper, 2013. Available at: `https://github.com/ethereum/wiki/wiki/White-Paper` (visited on 2015-08-03).

[19] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, 2014. Available at: `http://gavwood.com/Paper.pdf` (visited on 2015-09-24).

[20] ——, "Ethereum for dummies", Presentation, Ethereum Devcon1, 2015, Available at: `https://www.youtube.com/watch?v=U_LK0t_qaPo` (visited on 2016-02-08).

[21] A. Miller and J. LaViola, "Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin", 2014.

[22] V. Zamfir, "Challenges in public economic consensus", Presentation, Ethereum Devcon1, 2015, Available at: `https://www.youtube.com/watch?v=txJ4gXBCiYo` (visited on 2016-02-08).

[23] ——, "Reformalizing consensus", Draft, 2016.

[24] W. Diffie and M. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976, ISSN: 0018-9448. DOI: `10.1109/TIT.1976.1055638`.

[25] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism", *ACM Comput. Surv.*, vol. 17, no. 4, pp. 471–523, 1985, ISSN: 0360-0300. DOI: `10.1145/6041.6042`.

[26] P. Martin-Löf, "Constructive mathematics and computer programming", in *Proceedings Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, London, United Kingdom: Prentice-Hall, Inc., 1985, pp. 167–184, ISBN: 0-13-561465-1. Available at: `http://dl.acm.org/citation.cfm?id=3721.3731`.

[27] W. A. Howard, "The formulae-as-types notion of construction", 1980.

[28] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation", *Journal of Functional Programming*, vol. 23, pp. 552–593, 05 Sep. 2013, ISSN: 1469-7653. DOI: `10.1017/S095679681300018X`. Available at: `https://eb.host.cs.st-andrews.ac.uk/drafts/impldtp.pdf` (visited on 2016-02-13).

[29] *The effects tutorial*, The Idris community, 2015. Available at: `http://docs.idris-lang.org/en/latest/effects/index.html` (visited on 2015-10-26).

[30] A. Bauer and M. Pretnar, "Programming with algebraic effects and handlers", *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 108–123, 2015, ISSN: 2352-2208. DOI: `http://dx.doi.org/10.1016/j.jlamp.2014.02.001`. Available at: `http://math.andrej.com/wp-content/uploads/2012/03/eff.pdf`.

[31] A. Sabelfeld and A. C. Myers, "Language-based information-flow security", *IEEE journal on selected areas in communications*, vol. 21, no. 1, p. 2003, 2003.

[32] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically", in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14, San Diego, California, USA: ACM, 2014, pp. 411–423, ISBN: 978-1-4503-2544-8. DOI: `10.1145/2535838.2535851`. Available at: `http://doi.acm.org/10.1145/2535838.2535851`.

[33] V. Buterin. (2014). The latest EVM: "Ethereum is a trust-free closure system", Available at: `https://blog.ethereum.org/2014/03/20/the-latest-evm-ethereum-is-a-trust-free-closure-system/` (visited on 2016-02-13).

[34] C. Reitwiessner, *Solidity Documentation*, 2015. Available at: `http://solidity.readthedocs.org/en/latest/` (visited on 2016-02-08).

[35] G. Wood, *LLL PoC 6*, 2014. Available at: `https://github.com/chriseth/cpp-ethereum/wiki/LLL-PoC-6` (visited on 2016-02-13).

[36] *Serpent*, 2015. Available at: `https://github.com/ethereum/wiki/wiki/Serpent` (visited on 2016-02-13).

[37] E. Brady, "Cross-platform compilers for functional languages", *Under consideration for Trends in Functional Programming*, 2015. Available at: `https://eb.host.cs.st-andrews.ac.uk/drafts/compile-idris.pdf` (visited on 2015-12-18).

[38] ——, *Idris to PHP back end*. Available at: `https://github.com/edwinb/idris-php` (visited on 2016-02-08).

[39] *Idris 0.9.18 release notes*, 2015. Available at: `http://www.idris-lang.org/idris-0-9-18-released/` (visited on 2016-02-14).

[40] V. Buterin. (). Dependent types in Haskell programming, Available at: `https://wiki.haskell.org/Dependent_type#Dependent_types_in_Haskell_programming` (visited on 2016-02-26).

[41] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães, "Giving haskell a promotion", in *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, ACM, 2012, pp. 53–66.

[42] S. Lindley and C. McBride, "Hasochism: The pleasure and pain of dependently typed haskell programming", *ACM SIGPLAN Notices*, vol. 48, no. 12, pp. 81–92, 2014.

[43] V. Buterin, *EIP 101*, 2015. Available at: `https://github.com/ethereum/EIPs/issues/28` (visited on 2015-12-01).

[44] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs", *Software, IEEE*, vol. 25, no. 5, pp. 22–29, 2008.

[45] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software", in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ACM, 2007, pp. 1–8.

[46] J. G. Riecke, "Delimiting the scope of effects", in *Proceedings of the conference on Functional programming languages and computer architecture*, ACM, 1993, pp. 146–155.

[47] J. G. Riecke and R. Viswanathan, "Isolating side effects in sequential languages", in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: ACM, 1995, pp. 1–12, ISBN: 0-89791-692-1. DOI: `10.1145/199448.199450`. Available at: `http://doi.acm.org/10.1145/199448.199450`.

[48] J. Baeten, “A brief history of process algebra”, *Theoretical Computer Science*, vol. 335, no. 2–3, pp. 131–146, 2005, Process Algebra, ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/j.tcs.2004.07.036. Available at: http://www.sciencedirect.com/science/article/pii/S0304397505000307.

[49] L. Caires and H. T. Vieira, “SLMC: a tool for model checking concurrent systems against dynamical spatial logic specifications”, in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2012, pp. 485–491.

[50] A. Igarashi and N. Kobayashi, “A generic type system for the pi-calculus”, *Theoretical Computer Science*, vol. 311, no. 1–3, pp. 121–163, 2004, ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/S0304-3975(03)00325-6. Available at: http://www.sciencedirect.com/science/article/pii/S0304397503003256.

[51] The Ethereum foundation, *Solidity tutorials*, 2015. Available at: https://ethereumbuilders.gitbooks.io/guide/content/en/solidity_tutorials.html (visited on 2015-12-01).

A
Contract environment variables

Table A.1 shows the environment variables available during contract execution. Variable names differ by language; the ones shown below are from Solidity.

Name	**Data type**	**Description**
this	address	The current contracts address
<address>.balance	uint	Ether balance of a given address
block.coinbase	address	Current block miner's address
block.difficulty	uint	Current block difficulty
block.gaslimit	uint	Current block gaslimit
block.number	uint	Current block number
block.timestamp	uint	Current block timestamp
<block>.blockhash	bytearray	Hash of a given block
msg.data	bytearray	Complete calldata
msg.gas	uint	Remaining gas
msg.sender	address	Sender of current message (current call)
msg.value	uint	Ether amount sent with current message
tx.gasprice	uint	Gas price of current transaction
tx.origin	address	Sender of current transaction (full call chain)

Table A.1: Environment variables available during contract execution. Adapted from [51].