Towards Principled Compilation of Ethereum Smart Contracts (SoK)

Emilio Jesús Gallego Arias

Towards Principled Compilation of Ethereum Smart Contracts (SoK)

Emilio Jesús Gallego Arias
Centre de Recherche en Informatique
MINES ParisTech, PSL University — Blockchain Advanced Research & Technologies
Paris, France
e@x80.org

***Abstract*—A *blockchain* is a tamper-proof distributed transaction registry; first p opularized b y B itcoin [1], i t h as n ow been extended to support storage of arbitrary state and computations in-ledger. Ethereum [2] and its *smart contract* model have proven to be a very popular choice for this task, routinely managing assets valued in the billions. However, development of such contracts has been anything but easy. While formally specified, the Ethereum execution platform is based on a low-level machine, quite similar to assembly; semantics for contract operations such as *call* are quite complex, and the need for resource management creates unanticipated modes of failure. The dominant day-to-day programming platform for Ethereum is Solidity [3], an Object-Oriented language that identifies c ontracts w ith o bjects. While reasoning about Solidity programs is much easier than for their bytecode counterparts, it is not extent of challenges either, and moreover, Solidity lacks a source-level semantics, which forces developers to reason over output bytecode again. In this short paper we explore the main barriers to lift in order to achieve a principled compilation strategy for Solidity. We will review the standard concepts on verified a nd s ecure c ompilation, a nd frame them in the context of the Ethereum platform.**

***Index Terms*—secure compilation, formal methods, smart contracts, verification, Ethereum**

I. Introduction

Blockchain-based platforms need no introduction anymore; the tamper-proof distributed ledger model introduced by Bitcoin [1] has proven so far to be a hugely popular and robust platform for permission-less, trust-less transactions.

At its origins, Bitcoin was targeted to solve the "electronic cash problem". However, it also provided a "script" language, which allows to perform non-trivial computation as part of a transaction. A few years later, Ethereum [2] pushed this concept beyond popularizing the use of *smart contracts* in the distributed, permission-less ledger model. In this setting, a "smart contract" is a pair of mutable state and immutable code associated to an address, which is then run when input transactions are submitted to that particular code address.

The power of the model stems from three key points: first, contracts run in a general-purpose *virtual machine*, allowing arbitrary computation and providing a familiar programming model to developers by means of Solidity: an Object-Oriented programming language resembling JavaScript; second, there is no storage limitation for contracts other than read/write fees; and third, Ethereum provides consensus over the execution of the contract itself, which would require almost a majority of the miners to collude in order to alter code execution; this provides for great trust on how code will be run.

Typical use cases for smart contracts include management of fungible and non-fungible assets, voting, virtual markets, cryptographic commitment,... A remarkable success story has been the issuance of "Initial Coin Offerings", or participative, unregulated capital funding raising for Blockchain-based start-ups. In a ICO, a smart contract will manage a set of blockchain-issued shares of "tokens". Thanks to the ERC20 standard interface [4], "Token Exchanges" can interact with the contracts, allowing for an essential secondary token market where shares can be traded for fiat money or other crypto-assets.

More advanced standarized interfaces for contracts and new in-chain applications such as video games have been proposed [5], [6].

However, the success of smart contracts has also brought significant problematic in the form of critical software bugs. The nature itself of the contracts as asset managers means that bugs quickly translate to a loss of capital, and the immutability of the code means that mitigation may be hard or impossible. In many cases, the high enthusiasm over the new platform wasn't matched by the corresponding care when writing the asset management algorithms. Famous bugs such as the D.A.O. attack [7], [8] have made for losses in the tens of millions of dollars; and in the last year, many interesting papers have put focus on wide and serious class of problems present in deployed contracts.

We can split recent work on formal verification and analysis of Ethereum contracts in methods that work over the on-chain, low-level contract bytecode [9]–[18], and methods that focus on the functional specification of high-level, object-oriented Solidity code [19]–[23].

The first set does employ methods as diverse as rewriting, CFG, Datalog, decompilation, partial-order reduction, or symbolic execution; the methods targeting a high-level representation feature high-level type systems, program-logics, and state-transition diagrams.

However, the *gap* between high-level, programmer oriented Solidity code and low-level, Ethereum Virtual Machine is mostly unaddressed. As highlighted by Yoichi Hirai — who was for several years the head of verification at the Ethereum foundation: "the only way to know the meaning of a Solidity

program is to compile it into bytecodes". Indeed, Solidity code has no formal (or even informal) semantics, and compiler bugs have been common and in some cases very serious. In order to understand complex multi-contract interactions we are bound to directly look at the low-level VM code, which is indeed fully specified and thus able to provide truly formal guarantees.

Needless to state, being bound to the low-level semantics does pose some serious limitations: when end-to-end verification is required, complex and costly proofs about bytecode will have to be developed; rich functional properties such as [24], [25] are all but unapproachable this level. Even so, programmers will write and modify solidity code, making the proofs quickly stale. For people with less strict requirements, the use of defensive programming techniques and code analysis tools will provide for some relief, however, as contract code does grow in size, risk for the autonomous asset management does indeed increase.

In this short paper, we will discuss the challenges ahead to achieve a more principled compilation strategy for Solidity. In particular, we will describe the main tasks involved, and classify the required certified compilation technology in 3 categories: a) problems already solved by the state of the art in the general verified compilation setting; b) open problems in the general setting; and c) problems specific to the Ethereum / Blockchain architecture.

In the next section, we will briefly introduce the **Ethereum Virtual Machine (EVM)** and highlight its main points of interest from the optic of certified compilation, such as gas and exception semantics. After, we will survey the state of the art on certified and secure compilation. Next, we will explore the missing pieces to obtain a verifiable *trace preservation* property between the high-level and low-level semantics of Solidity / EVM; however — as we will see — in the presence of adversarial code semantic preservation may not be enough, and completeness with regards to low-level contexts may be required. We conclude with perspectives and future work.

II. Solidity and the Ethereum Virtual Machine

The Ethereum [2] network is based on the same principles than Bitcoin: blocks are produced at a *quasi*-constant rate, and they contain a pointer to their parent plus a set of cryptographically signed transactions. *Miners* (or validators) compete to generate the next block, which will be awarded by a random lottery but proportional to the raw computing power of each validator. The validator that *wins* the privilege to produce the next block will get a reward in the form of *Ether*, and there is incentive for the rest of validators to follow the longest-chain, or else waste large amounts of electric power. For a transaction already included in a chain, the longest is the suffix after it, the most likely is that it is valid, as this means large amounts of energy were used to validate that chain.

For the rest of the paper, we assume that the "global state" of the chain is encoded by a function $\sigma : \mathsf{address} \rightarrow \mathsf{acc\_state}$. Addresses are 160-bit identifiers, and the relevant bits of each account state are $\sigma[a]_s$, the *storage* for a, a map between 256-bit integer values, the *balance* for a, written as $\sigma[a]_b$, which denotes the amount of Ether that particular account has; and (optionally) $\sigma[a]_c$: the code of the account in the from of EVM opcodes. If an account a has some code attached to it we say it is a "contract", otherwise we refer to it as a "regular" account.

The fundamental state-altering operation in Ethereum is the *transaction*. A transaction $T = (n, p, g, t, v, \{i, d\})$ consists of a *nonce* n, a *gas* price p, and limit g, an addressee t, an Ether amount v to send to t, and either an *initial contract code* i, used if $T_t = \emptyset$ or some *call data* T_d, which will be the input for the code in $\sigma[t]_c$ otherwise. We omit the transaction signature fields here, but they are essential of course to verify that the transaction is authorized by the proper account.

Transactions are signed and submitted by users to a pool, then selected by validators to be included into a block; from an external point of view, transaction execution can be modeled as a function $\sigma' = \Upsilon(\sigma, T)$, where Υ is in charge of executing the transaction. A valid transaction will always modify the state, even if just to record failure by incrementing the nonce of the submitting account, and subtracting gas if possible. A complete description of transaction processing is impossible due to lack of space, but basically, contract code will be executed by the Ethereum Virtual Machine, which is a stack-based machine with access to local memory and the global state, as long as it is authorized to do so. Validators are expected to execute contract bytecode according to the official specification, and execution is deterministic. We highlight 3 relevant aspects:

- the most interesting (and complex) operations are contract *creation*, *call*, and *destruction*. Indeed, the EVM allows complex interactions among contracts in a single transaction, including re-entrancy,
- a transaction may fail due to a variety of reasons: running out of *gas*, exceeding the *call-depth limit*, or by executing invalid code. Failure of a transaction does imply a revert of the global state except for the fees,
- failure of a transaction is not the same than failure of a contract call; indeed, a call to another contract may fail, however the caller's execution may continue and recover from the failed call.

The bytecode language provided by the EVM is pretty rich; we can query almost any parameter of the current transaction, read code, hash values, delegate current permissions, etc...

Solidity

As mentioned before, low-level EVM bytecode does not make for an ideal smart contract programming language: it lacks compound types, names, subroutines, and writing efficient code is hard due to the nature of the stack machine. To alleviate these problems, *Solidity* was introduced early on. Solidity is based on an Object-Oriented paradigm, identifying classes with contract specifications, and object instances with on-chain contracts. A typical Solidity contract will have a *global storage* section, which is the state that will be persisted between *method* calls, and methods which are the entry point to the contract from transactions. For example a (buggy) counter is implemented as:

```
contract Counter {
 uint public ctr = 0;
 function incr(uint value) public { ctr += value; }
}
```

compare this 2 lines with the 100s of instructions the contract is compiled to. Given a contract of type `Counter` at address a, we can then call `a.incr(value)` and set the counter to any arbitrary value we would like it to be.

As of today, Solidity is quite feature-rich and includes some defensive programming facilities based on past experience and problems; it also inherits some of the oddities of the underlying low-level programming model. For example, contracts do have a default action, which is triggered when an Ether-only transaction is sent; now, Solidity forces the programmer to be explicit about this, otherwise rejecting unexpected transactions right away.

III. Notes on Verified and Secure Compilation

Before distilling what would a secure compilation chain mean for Solidity + EVM, we briefly recall some basic concepts from the verified and secure compilation literature. We mostly follow the excellent presentation of [26], which provides an in-depth overview of the current state of the art. We assume a *source* language S, and a compilation procedure $\downarrow$ that will transform the source program $\mathsf{P} \in \mathsf{S}$ to a program $\mathsf{P}\downarrow$ in the target language T.

The goal is to show that the compilation procedure respects semantics, which we can write as $\mathsf{P} \approx \mathsf{P}\downarrow$ where $\approx$ is some desired equivalence. Given that the source and target languages are different, it is customary to use indeed a notion of *observation* between the programs. We say that $t \in \pi$ is a trace of a program execution; we write $\mathsf{P} \rightsquigarrow t$ for "program P produces trace t when executed under the source semantics", and similarly $\mathsf{P} \rightsquigarrow t$ for the target semantics. Traces may be finite or infinite, but the key point is they allow us to relate behaviors from the source and target worlds, each one totally different languages and execution rules. Armed with these definitions, we can define now:

Proposition 1 (Soundness of Compilation):

$$\forall \mathsf{P}, t \in \pi,\ \mathsf{P} \rightsquigarrow t \Rightarrow \mathsf{P}\downarrow \rightsquigarrow t$$

the above theorem basically states that for every possible trace in the source language, its compiled version will have the same trace; note that this definition doesn't forbid the target translation from having more behaviors than the original program. More details can be found in seminal works [27]–[29]; there are many subtle variations of the idea, and in for each case, details can greatly differ.

Soundness (and its variations) do help a lot when we control all the code that we will run, bringing us to a full correctness theorem for our compiler. However, in open scenarios such guarantees may not be enough, take for example the case of linking against untrusted code. Following the presentation of [26] again, we introduce the notion of a *context* $C_{\mathsf{S}}[\cdot]$; a context represents the environment (including code, inputs, etc...) in which a program is executed, we write $C_{\mathsf{S}}[\mathsf{P}]$ for the composition of a program with its context. *Secure* compilation assumes that the *target context* is in the hands of the adversary, thus in order to soundly reason in the source language, it should be the case that the source contexts can capture all the possible behaviors when the target code is run:

Proposition 2 (Robust Trace Conservation):

$$\forall \mathsf{P}, C_{\mathsf{T}}, t,\ C_{\mathsf{T}}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow \exists C_{\mathsf{S}},\ C_{\mathsf{S}}[\mathsf{P}] \rightsquigarrow t$$

Proving robust trace preservation is in general very challenging; one example technique is given in [30], where thanks to an universal embedding of target contexts, we can always back-translate making RTC hold.

We end this section recalling the restrictions on the class of preserved traces is quite common; both in the general literature, and in Ethereum-specific analysis tools. In particular, two standard criteria are:

- *Safety properties*: some bad state is never reachable.
- *Liveness properties*: some good state is always reachable.

IV. Ethereum: A Case Study

Armed with the formal machinery that we have developed in the previous sections, we proceed to study its applicability for the Solidity / EVM pair. We first try to define traces, then we move to contexts. As it often happens with formal proofs, to find the right definitions turns out to make for the large majority of the proof effort.

A. Traces and Soundness

In order for correctness [and completeness] to be properly defined, we must precisely distinguish two traces t_1, t_2 when they do yield different global states. In particular, let σ_1, σ_2 be the global (or blockchain state) associated to t_1, and t_2 respectively; it must then hold that:

$$\sigma_1 \neq \sigma_2 \Rightarrow t_1 \neq t_2$$

Ideally, we would like the other direction to hold too, but it is much harder to achieve.

Let's revisit for a moment the definition of trace in Grishchenko *et al.* In their case, a trace is "a sequence of action, where each action consists of an opcode, the address of the executing contract, and a sequence of arguments to the opcode."

Unfortunately that definition of trace is not adequate for our source language, Solidity in this case. Traces at the opcode level are too fine-grained for a source language based on object. Take for example the semantics of [22], based on the standard one of [31], which do consider method call a primitive operation, as opposed to the EVM.

While we could use a heterogeneous trace relation, things would become very complicated quickly. Instead, we may try to follow the approach of [29], which was able to obtain great success when formally relating C code to low-level assembly. In this setting, traces are order-preserving sequences of effect, usually read/writes from the global state.

This definition of trace does get us much closer to a sensible theorem statement. A "large" step in the source language, such

as a method call, should easily match the trace of the compiled target. So far our use case seems to be reasonably standard, but unfortunately, we are not there yet. The reason is *gas*. As we briefly hinted in the previous sections, each transaction does come with a *gas* limit, that is to say, an upper bound on the number of opcodes the EVM will execute before throwing a fatal "out-of-gas" exception. Semantics of gas become quite complicated when a multi-contract call is present, as an out-of-gas exception in a called contract will not be fatal for the caller [but only exhaust its assigned budget]. This is already the source of many problems, even when considering only bytecode [18], but in our case, imposes either a painful gas precondition on the statement of the theorem, or will require to use some new technique to relate instruction cost among the target and source traces. In a sense gas consumption can be seen as an "effect", which treated in the naïve way quickly produces unmanageable traces and proofs. The core of the problem here is due to gas not being about asymptotic cost, but about *exact* cost execution. A good example on why this is a hard problem is that adding a trivially sound optimization to our compiler will suddenly make the target language not fail anymore, breaking preservation. Recent interesting work in this area is [32], which indeed targets a typed version of the EVM, and uses execution bounds to guarantee soundness.

B. Security vs Compartmentalization

We briefly discuss execution contexts, and how adversarial EVM code could compromise an otherwise correct program. In the case of Solidity, it "source contexts", C_S do correspond to the "execution environments" that appear in regular OO-semantics; already with some caveats with regards to object references (see [22]). However, at the bytecode level, it is not clear that low-level code can behave "worse" than source contexts, other than previously discussed gas or recursion problems. While it seems to us that the current compilation scheme may not indeed be secure, a reasonable amount of hardening should allow us to recover a security proof.

However, the line we are walking seems pretty fine here; when the source-level semantics hold a pointer to a different contract, some assumptions on well-formedness, or even on behavior can occur. For example the `payable` attribute on a contract type would allow the compiler to perform some optimizations that are unsound under the perspective of secure compilation. As discussed in [26], Sec. 3, we could instead re-formulate our problem as a compartmentalization problem where several components with given interfaces can become compromised and thus behave arbitrarily bad. A general treatment of these systems seems like an open problem as of today; however in our opinion work in that area is highly relevant smart contracts with an "open" formulation.

V. Perspectives and Future Work

We hope to have shed some light on the challenges that attack-resistant compilation of contracts does entail, and we hope this information to be useful for the language and run-time designers of platforms amenable to secure computing.

In the smart-contract world, some have called for validators to directly implement higher-level languages. I do agree that this would be a welcome step, lessening the need for complex compilation chains and proofs such as the ones discussed here.

However, reality seems to disagree and the next version of Ethereum and other smart contract platforms will be likely based on low-level languages such as WebAssembly. It seems that some inherent conflict stems between validators and contract writers, as they disagree on the layer where complexity should be pushed; we then expect that verified compilation techniques will be relevant for the platforms still in the medium term.

References

[1] S. Nakamoto, *Bitcoin: A Peer-to-Peer electronic cash system*.

[2] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*.

[3] G. Wood, C. Reitwiessner, A. Beregszaszi, Y. Hirai, *et al.* (2019). The solidity contract-oriented programming language, [Online]. Available: https : / / github . com / ethereum/solidity.

[4] V. Buterin and F. Vogelsteller. (2017). Erc-20 token standard, [Online]. Available: https : / / github . com / ethereum/EIPs/blob/master/EIPS/eip-20.md.

[5] W. Entriken, D. Shirley, J. Evans, and N. Sachs. (2017). Erc-721: Non-fungible token standard, [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[6] D. Eigenmann and J. Izquierdo, *Erc900: Simple staking interface*.

[7] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust: 6th International Conference, POST 2017*, M. Maffei and M. Ryan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8.

[8] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," *CoRR*, vol. abs/1703.06322, 2017.

[9] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 254–269. DOI: 10.1145/2976749.2978309.

[10] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds., Cham: Springer International Publishing, 2018, pp. 243–269.

[11] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. M. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: a complete formal semantics of the ethereum virtual machine," in *31st IEEE Computer Security Foundations Symposium, CSF*

2018, Oxford, United Kingdom, July 9-12, 2018, IEEE Computer Society, 2018, pp. 204–217. DOI: 10.1109/CSF.2018.00022.

[12] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, S. K. Lahiri and C. Wang, Eds., ser. Lecture Notes in Computer Science, vol. 11138, Springer, 2018, pp. 513–520. DOI: 10.1007/978-3-030-01090-4_30.

[13] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: ACM, 2018, pp. 67–82. DOI: 10.1145/3243734.3243780.

[14] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18, San Juan, PR, USA: ACM, 2018, pp. 653–663. DOI: 10.1145/3274694.3274743.

[15] I. Sergey, A. Kumar, and A. Hobor, "Temporal properties of smart contracts," in *8th International Symposium, ISoLA 2018*, T. Margaria and B. Steffen, Eds., ser. Lecture Notes in Computer Science, vol. 11247, Springer, 2018, pp. 323–338. DOI: 10.1007/978-3-030-03427-6_25.

[16] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "Scompile: Critical path identification and analysis for smart contracts," *CoRR*, vol. abs/1808.00624, 2018. arXiv: 1808.00624.

[17] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *CoRR*, vol. abs/1809.03981, 2018. arXiv: 1809.03981.

[18] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 116:1–116:27, 2018. DOI: 10.1145/3276486.

[19] Z. Yang and H. Lei, "Lolisa: Formal syntax and semantics for a subset of the solidity programming language," *CoRR*, vol. abs/1803.09885, 2018. arXiv: 1803.09885.

[20] Z. Yang and H. Lei, "Fether: An extensible definitional interpreter for smart-contract verifications in coq," *IEEE Access*, vol. 7, pp. 37 770–37 791, 2019. DOI: 10.1109/ACCESS.2019.2905428.

[21] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '18, Markham, Ontario, Canada: IBM Corp., 2018, pp. 103–113.

[22] S. Crafa, M. D. Pirro, and E. Zucca, "Is solidity solid enough?" In *3rd Workshop on Trusted Smart Contracts*, 2019.

[23] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," vol. abs/1901.01292, 2019. arXiv: 1901.01292.

[24] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub, "Higher-order approximate relational refinement types for mechanism design and differential privacy," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, Mumbai, India: ACM, 2015, pp. 55–68. DOI: 10.1145/2676726.2677000.

[25] ——, "Computer-aided verification in mechanism design," in *Web and Internet Economics: 12th International Conference, WINE 2016, Montreal, Canada, December 11-14, 2016, Proceedings*, Y. Cai and A. Vetta, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, Dec. 2016, pp. 279–293. DOI: 10.1007/978-3-662-54110-4_20.

[26] C. Hritcu, *The Quest for Formally Secure Compartmentalizing Compilation*. 2019.

[27] G. C. Necula, "Proof-carrying code," in *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, P. Lee, F. Henglein, and N. D. Jones, Eds., ACM Press, 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[28] A. W. Appel, "Foundational proof-carrying code," in *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, IEEE Computer Society, 2001, pp. 247–256. DOI: 10.1109/LICS.2001.932501.

[29] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009. DOI: 10.1145/1538788.1538814.

[30] M. S. New, W. J. Bowman, and A. Ahmed, "Fully abstract compilation via universal embedding," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, J. Garrigue, G. Keller, and E. Sumii, Eds., ACM, 2016, pp. 103–116. DOI: 10.1145/2951913.2951941.

[31] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight java: A minimal core calculus for java and gj," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, 2001. DOI: 10.1145/503502.503505.

[32] P. Wang, "Type system for resource bounds with type-preserving compilation," PhD thesis, Massachusetts Institute of Technology, 2019.