Bridging Specification and Implementation in Smart Contract Languages

Cas van der Rest
IOHK, The Netherlands http://www.casvanderrest.nl/

Abstract

Before entrusting a smart contract with our funds or data, it is essential to fully understand its terms. A formal specification of the smart contract language's semantics is crucial for expressing these terms unambiguously. However, before a specification can serve as a reliable tool for assessing a contract's behavior, we must first establish a strong connection between the specification and implementation, a task that is further complicated by the rapid evolution of smart contract languages and the common prioritization of implementation over formal specification during (early) development.

In this paper, we propose a strategy to prevent divergence between implementation and specification by transpiling Nanopass intermediate representations (IRs) into mutually defined families of inductive data types in Agda. We enforce completeness of typing relations using Agda's dependent type system and meta-programming capabilities. A key outcome of our approach is that any meaningful syntactic change—such as the addition or removal of production rules or non-terminals—results in a compile-time error in the specification. Although our approach was developed in the context of Compact—the smart contract language of the Midnight blockchain—we believe that it may serve as a general template for synchronization between Agda-based specifications and smart contract language implementations.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

Acknowledgements I want to thank ...

1 Introduction

Smart contracts potentially handle large amounts of funds, and as such it is crucial precisely understand its terms, as well as trust that a system will faithfully execute these terms. Failing to understand the terms of a smart contract may lead to substantial financial losses, as exemplified by the infamous exploit of the TheDAO contract [8] in 2016. Subtleties around the expression of control flow in Solidity smart contracts allowed hackers to launch a reentry attack and capture the equivalent of $50 in Ether, resulting in hard fork of the Ethereum blockchain to recover the stolen funds.

This exploit perfectly illustrates the need to be able to unambiguously express the terms of a smart contract. A formal mathematical *specification* of the smart contract language vital for expressing a contract's intended semantics. However, the utility of formal specifications in assessing a contract's validity hinges on whether it faithfully reflects how smart contracts are executed by the *implementation* of a system. This problem of how to bridge the gap between specification and implementation is an age-old question in the field of programming languages, resulting in many different approaches to tackle the problem, ranging from fully-verified compilers [4, 5] to more light-weight approaches such as certifying compilation [6] or conformance testing.

While previous work offers us a plethora of techniques for connecting specification and implementation, additional challenges arise when they are applied outside the context of academic research. Typically, smart contract languages are developed in an environment

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:7
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that is characterized by rapid evolution of the language's design as well as prioritization of implementation over the development of (formal) specifications. This presents a dilemma: do we develop the specification *with* the language, embracing the additional maintenance costs incurred by an evolving design, or do we wait for the language's design and implementation to stabilize. Although waiting appears tempting, it is important to highlight that there are downsides too. That is, we lose potential some synergy between specification and implementation, where by forcing ourselves to express mathematically what (we think) we are doing, we may uncover flaws in our thinking and design. This allows a specification to inform the design of the very language it is specifying. It becomes much harder to backtrack on mistakes when strictly sequencing the development of a language's implementation and specification.

Clearly, the ideal scenario would be to develop a specification in conjunction with an implementation, while maintaining a strong connection between the two. However, if we require implementation and specification to be connected through formal proofs, e.g., by having a verified compiler, this becomes entirely infeasible for almost all projects due to the tremendous amount of resources required. This begs the question of whether we can settle for a more light-weight approach where we develop the implementation and specification simultaneously but separately, connecting them in a way that prevents divergence in the presence of design changes but does not impede the development process by requiring proofs to be written before code reaches production.

In this paper, we present such a light-weight strategy for bridging the specification and implementation of smart contract languages. In short, our approach works by transpiling the syntax definition of Nanopass [3] intermediate representations (IRs) to mutually-defined families of inductive data types in Agda [7], leveraging its dependent type system and meta-programming capabilities to enforce completeness of typing rules defined over the transpiled syntax. This approach has two important benefits. First, it guards against divergence of the specification and implementation, by using the compiler's internal syntax definition as the source of truth for the languages abstract syntax. This ensures that any meaningful syntactic change to the languages abstract syntax—in the form of adding, removing or changing production rules or non-terminals—manifests as a compile time error when type checking the specification. As a result, we can fully automatically check whether the specification and implementation are synchronized. The second important benefit of our approach is that it does not impact the compiler development cycle in any way. By repurposing the definition of Nanopass IRs as a DSL for specifying the language's abstract syntax, we integrate the specification and implementation in a way that neither induces any additional overhead in the compiler nor does it induce additional maintenance work for compiler developers.

More concretely, this paper is structured as follows:

- In Section 2, we illustrate, by example, how one would define and formally specify a smart contract language using our approach.
- In Section 3, we present an approach for leveraging Agda's dependent type system and meta-programming capabilities to statically enforce completeness of typing relations defined over the transpiled syntax.

Finally, we conclude and discuss future work in Section 4.

1.1 Industrial Application

The techniques in this paper have been developed and applied in the context *Compact* [1], the smart contract language of the Midnight blockchain [2]. In that context, we successfully

```
(define-language Lstlc        -- Declares the name of the language
  (terminals                  -- Declares the terminals of the languag
   (string (name))
   (number (nat)))
  (Type (type)                -- Declares "Type" and its meta-variable
    (tbool)
    (tnat)
    (tfun type1 type2))
  (Expr (expr)                -- Declares "Expr" and its meta-variable
    (evar name)
    (etrue)
    (efalse)
    (elit nat)
    (eif expr1 expr2 expr3)
    (elam name expr1)
    (eapp expr1 expr2)
    (eadd expr1 expr2)))
```

Figure 1 Nanopass IR definition for a Simply-Typed λ-calculus

applied the techniques described in this paper to develop an Agda specification of Compact's static semantics next to its Nanopass implementation. While Compact is not a particularly large language, its abstract syntax still contains in the order of 100 production rules spanning several tens of non-terminals. By extension, the formal specification of its type system contains an equal number of typing rules respectively judgments. Using the techniques described in this paper, we managed to keep this formal specification from diverging from its implementation, despite the rapid pace at which the language is currently evolving. We choose to present them here in a more general setting to illustrate their potential application outside the context in which they were initially developed.

2 Integrating Specification and Implementation, by Example

In this section we demonstrate the process for defining, transpiling, and specifying a language definition. For this purpose, we show, as a three-step process, how to formally specify the static semantics of a simply-typed λ-calculus.

2.1 Step 1: Declare Abstract Syntax of the Target Language as a Nanopass IR

We start by declaring the abstract syntax of our language as a Nanopass IR. This definition of the abstract syntax is regarded as the source of truth, and is used both by the implementation and specification. Figure 1 shows the definition of a Nanopass IR for a simply typed λ-calculus. Along with some metadata defining the name of the IR (`Lstlc`), it declares two non-terminals: `Type`, defining the syntax of types, and `Expr`, defining the abstract syntax of terms. For each non-terminal, we must also declare a *meta-variable* (respectively `type` and `expr` for types and terms of the language), which are used to refer to their associated non-terminal. Meta-variables also play a crucial role in transpilation; when converting the untyped Nanopass IR to a typed Agda definition, the required type information is recoverd by resolving these meta-variables.

```
mutual
  data Expr : Set where
    evar : String → Expr
    etrue : Expr
    efalse : Expr
    enum : ℕ → Expr
    eif : Expr → Expr → Expr → Expr
    elam : String → Expr → Expr
    eapp : Expr → Expr → Expr
    eadd : Expr → Expr → Expr

  variable
    expr expr₁ expr₂ expr₃ expr' : Expr

  data Type : Set where
    tbool : Type
    tnat : Type
    tfun : Type → Type → Type

  variable
    type type₁ type₂ type₃ type' : Type
```

Figure 2 Transpiled Agda definition of the Nanopass IR definition shown in Figure 1.

2.2 Step 2: Transpilation of Abstract Syntax to Agda

The next step is to transpile the Nanopass IR defined in Figure 1 to Agda. This generates a mutually-defined family of inductive data types. Figure 2 shows the generated Agda definition. Each non-terminal in the abstract syntax corresponds to a data type, and each production rule to a constructor. To make sure that the generated type signatues are accepted by Agda, we must resolve meta-variables referring to both terminals and non-terminals. For example, we transpile the production rule for λ-abstraction as follows:

`(elam name expr1)` $\mapsto$ elam : String → Expr → Expr

Here, to ensure that the generated type signature is accepted by Agda, we must resolve the meta-variable `expr1` to the Expr data type, and the meta-variable `name` to the String type.

In addition to a mutually-defined family of inductive types, transpilation also generates several *generalized variables* for each data type, indicated by variable. These serve a similar purpose to meta-variables in the Nanopass IR, in that they are used to refer to universally-quantified values of their corresponding types.

2.3 Step 3: Define Typing Rules as an Inductive Relation over Abstract Syntax

Now that the abstract syntax of the target language is available in Agda, we can define the type system. Typically, one does this in Agda by declaring an inductive relation over terms (or, term-indexed data type), whose constructors the typing rules, i.e., the different ways in

```
⊢-evar :  name ↦ type ∈ Γ
          ────────────
          Γ ⊢ evar name : type

⊢eif   :  Γ ⊢ expr₁ : tbool
        → Γ ⊢ expr₂ : type
        → Γ ⊢ expr₃ : type
          ────────────────────
          Γ ⊢ eif expr₁ expr₂ expr₃ : type

⊢eapp :  Γ ⊢ expr₁ : tfun type₁ type₂
       → Γ ⊢ expr₂ : type₁
          ──────────────────
          Γ ⊢ eapp expr₁ expr₂ : type₂
```

Figure 3 Excerpt of the typing rules for the simply typed λ-calculus defined in Figures 1 and 2, defined as constructors of an inductive relation over context, term, and type.

which a proof of well-typedness can be constructed. Generally, this relation has additional positions for tracking type information and contextual information. In this case, we use a three-place relation over context, terms, and types:

```
data _⊢_:_ (Γ : Context) : Expr → Type → Set where
```

Figure 3 shows an excerpt of the definition of the static semantics for the target language, by declaring constructors corresponding to the typing rules for variables, if-then-else expressions, and function application.

3 Ensuring Completeness of Typing Relations

Right now, a compile-time error will be triggered in the following cases:

- a non-terminal is removed,
- a production rule is removed, or
- a production rule is changed.

In summary, by defining our formal specification on top of the transpiled syntax, we are guaranteed that our specification can only refer to syntactic elements that are part of the compiler's internal definition. What is explicitly not guaranteed is that the specification is a *complete* one. That is, the following changes to the language will not trigger a compile-time error:

- a non-terminal is added, or
- a production rule is added.

More likely than not, such syntactic changes signify the addition of a new feature to the language, for which we should also extend the formal specification. To ensure that the addition of new syntax to the language triggers a compile-time error, we must perform additional checks on the Agda side. Specifically, we must (1) declare which data types make up the language, (2) check that we have a corresponding typing relation for each data type in the language's syntax, and (3) check that for every constructor there is a corresponding typing rule in the associated typing relation.

For (1), the transpilation tool generates an additional definition that collects the non-terminals of the language together with their meta-variables:

```
Lstlc : List (Set × String)
Lstlc = (Expr , "expr") :: (Type , "type") :: []
```

To ensure that each non-terminal has an associated typing relation, we declare an instance of the HasTyping for the Lstlc language. This instance contains a proof witnessing that every non-terminal of the language has an associated typing relation in the form of a three-place relation over context, terms, and types.

```
Typing S = ∃₂ λ (CTX : Set) (I : CTX → Set) → (ctx : CTX) → S → I ctx → Set
record HasTyping (syn : List (Set × String)) : Set₁ where
  field rels : All (Typing ∘ proj₁) syn
```

As a result, by defining an instance of the form HasTyping Lstlc, we are forced to declare a typing relation for each non-terminal. Furthermore, whenever a new non-terminal is added in the Nanopass IR, after transpilation of the updated syntax, this instance becomes ill-typed, and we are must declare a new typing relation for the newly added non-terminal.

Finally, we use Agda's meta-programming capabilities to check that the declared typing relations cover every production rule of the syntax. In short, this check is performed by asserting that for every constructor in the untyped syntax, there is a constructor of the corresponding typing relation that has that constructor in the term position (i.e., S in the definition of Typing). We invoke this check for Lstlc as follows:

```
unquoteDecl = checkRels (getTyping Lstlc) []
```

If the typing rules shown in Figure 2 were the only rules we defined, invoking the meta-program above would result in the following type error when checking the specification, indicating which relations are missing which typing rules.

```
Discovered missing rule(s) while checking coverage of relation _⊢_: _
  ---> No typing rule found for constructor etrue
  ---> No typing rule found for constructor efalse
  ---> No typing rule found for constructor enum
  ---> No typing rule found for constructor elam
  ---> No typing rule found for constructor eadd
```

4 Conclusion and Future Work

In this paper, we presented an approach for connecting the specification and implementation of smart contract languages through the transpilation of Nanopass IRs. While our approach was developed and applied in the context of the Compact language, its principles and methodology are generalizable to other (smart contract) languages.

For future work, we aim to further enhance the accessibility and applicability of our approach by making our tools publicly available as open-source projects, and leveraging our extraction mechanism to enable further compiler verification. For example, we could explore certification [6] techniques to formally verify compiler correctness.

References

1. URL: `https://docs.midnight.network/develop/reference/compact/`.
2. URL: `https://midnight.network/`.
3. Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 343–350. ACM, 2013. `doi:10.1145/2500365.2500618`.
4. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014. `doi:10.1145/2535838.2535841`.
5. Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.
6. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 333–344. ACM, 1998. `doi:10.1145/277650.277752`.
7. Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. `doi:10.1007/978-3-642-04652-0\_5`.
8. Cryptopedia Staff. The dao: What was the dao hack?, Oct 2023. URL: `https://www.gemini.com/cryptopedia/the-dao-hack-makerdao`.