Semântica Operacional Executável para Michelson em Why3*

Luís Pedro Arrojado Horta[1], Mário Pereira[2], and Simão Melo de Sousa[1]

[1] Universidade da Beira Interior, Rua Marquês D'Ávila e Bolama 6201-001 Covilhã, Portugal
[2] FCT-UNL, NOVA - LINCS

Resumo Neste artigo apresentamos uma semântica operacional executável de uma linguagem de *smart contracts* em Why3. Esta implementação representa o primeiro esforço para a criação de uma plataforma de verificação de *smart contracts* para a *blockchain* Tezos. Esta plataforma visa validar não só o mecanismo de execução dos mesmos, bem como os *smart contracts* em si. Descrevemos aqui um interpretador para Michelson *big-step* que além de ser executável, queremos também que seja uma semântica formal da nossa lógica, isto é, uma função matemática que a qualquer programa Michelson atribui o seu valor semântico. Assim sendo, é necessário mostrar que `eval` é uma função total, o que equivale a provar a sua terminação. Graças ao mecanismo de extração de código do Why3, é possível obter uma implementação OCaml deste interpretador.

Keywords: Operational Semantics · Formal Specification · Michelson · Smart Contracts · Tezos · Why3.

1 Introdução

Um sistema de *blockchain* [8], funciona como um livro-razão onde ficam registadas todas as transações feitas entre todos os diversos utilizadores de uma determinada *blockchain*. Com o aparecimento desta tecnologia, surgiram uma série de possibilidades que poderiam advir da aplicação deste sistema a diversas áreas que nada aparentam ter em comum com a informática, como por exemplo a política ou a medicina [5].

Neste artigo focamo-nos na *blockchain* Tezos [6], mais propriamente na linguagem para escrita de *smart contracts*, denominada de Michelson. Esta linguagem funciona através da reescrita de uma pilha e apresenta um paradigma funcional, o que a torna especialmente atrativa no que diz respeito à utilização de métodos formais para a sua verificação.

Os *smart contracts* são programas escritos numa linguagem específica e posteriormente executados numa *blockchain*. Considerando que os outorgantes destes contratos não confiam uns nos outros e tendo em consideração que grande parte

* Investigação financiada pela Tezos Foundation no âmbito do projecto FRESCO - FoRmal vErification of Smart COntracts.

destes contratos envolvem a transferência de fundos com valor monetário (*tokens*) entre utilizadores, torna-se evidente a necessidade de criar uma plataforma de verificação formal dos mesmos. Por forma a atingir esse objetivo, decidimos que a ferramenta de verificação dedutiva Why3 [4] seria a base de criação da referida plataforma.

Este artigo estará então organizado da seguinte forma: a secção 2 – **Trabalho Relacionado** – descreve brevemente algum trabalho já executado por outros autores relacionado com esta temática. Na secção 3 – **Semântica Operacional Executável** – apresentamos ao leitor alguns detalhes da especificação da linguagem Michelson em Why3, bem como partes da implementação função *eval* em WhyML. Finalmente, a secção 4 – **Conclusões e Trabalho Futuro** – contém um resumo das principais conclusões deste projeto, assim como uma breve discussão do trabalho futuro.

2 Trabalho Relacionado

O termo *software bug* é uma constante no mundo da programação. Estes *bugs* resultam em comportamento não previsto na execução de determinada porção de código, o que pode provocar perdas graves, que podem ser monetárias ou até mesmo vidas humanas [7]. A origem de um *bug* nem sempre é clara. Algumas das suas causas mais comuns são, a introdução do mesmo de forma não intencional por parte do programador, erros de tipagem, ou até mesmo falhas no desenho da linguagem de programação em si.

No que diz respeito à verificação formal de *smart contracts* já existem alguns esforços para criar plataformas de verificação para os mesmos, como é o caso de *Nehai, Z. e Bobot, F.* que no trabalho descrito em [9] utilizam Why3 para escrever contratos para a *blockchain* Ethereum [3]. Também, *Bhargavan, K., et al.* desenvolveram uma *framework* para analisar e verificar a correção funcional de *smart contracts* Ethereum por tradução para F^* [2]. Ainda para a mesma *blockchain*, *Abdellatif, T.* e *Brousmiche, K.* utilizaram a *framework* BIP [1] para modelar e verificar os referidos contratos utilizando *statistical model checking*. Utilizando o Coq Proof Assistant, *Zheng Yang* e *Hang Lei* desenvolveram um motor de verificação e prova denominado FEther que combina execução simbólica com prova de teoremas de lógica de ordem superior no seu trabalho apresentado em [14].

Como referido na secção anterior, neste trabalho o nosso foco são os *smart contracts* escritos em Michelson para a *blockchain* Tezos. A empresa Nomadic Labs, formalizou uma semântica para Michelson utilizando o Coq Proof Assistant [11]. Este trabalho difere do nosso, pois o nosso objetivo reside na automação da verificação para que possa ser transparente ao utilizador o uso da plataforma em Why3; ao contrário do Coq, no qual a prova é feita manualmente.

3 Semântica Operacional Executável

Nesta secção apresentamos ao leitor alguns dos principais detalhes da implementação da semântica operacional *big-step* Michelson em Why3. Esta implementação foi baseada na especificação de Michelson feita pela Tezos Foundation na sua página web [13]. A restante parte desta secção encontra-se estruturada da seguinte forma: a subsecção 3.1 realça algumas das escolhas levadas a cabo durante a especificação da linguagem Michelson em Why3; a subsecção 3.2 mostra diversos detalhes da função `eval`; enquanto que a subsecção 3.3 faz o paralelo entre o código OCaml extraído do Why3 com o apresentado na subsecção anterior.

3.1 Especificação Michelson em Why3

Na linguagem Michelson existem 4 tipos de dados primitivos para constantes, nomeadamente `string`, `int`, `nat` e `bytes`. Temos ainda os tipos `bool` para booleanos, `Option` τ para um valor v opcional do tipo τ, denotado por `None` ou `Some` v, e ainda o tipo `unit`. Para uma maior facilidade de leitura, a tabela 1 mostra a correspondência entre os tipos Michelson e os tipos Why3.

Tipo Primitivo Michelson	**Tipo Correspondente Why3**
string	seq char
nat	nat
int	int
bytes	seq bv.BV8
bool	bool
option τ	option τ
unit	unit

Tabela 1. Correspondência entre tipos primitivos Michelson em Why3.

Ambos os tipos `int` (para números inteiros) e `nat` (para números naturais), são de precisão arbitrária em Michelson. No caso do Why3 o tipo `int` com precisão arbitrária não representa qualquer problema, pois é exatamente essa a sua definição. Já o tipo `nat` não existe naturalmente, e desta forma, definimos o mesmo como mostra a figura 1.

```
type nat =  O  | S  nat
```

Figura 1. Tipo `nat` definido em Why3.

As operações entre constantes do tipo `nat` foram implementadas utilizando funções recursivas, com a ressalva de que a operação de divisão tem como tipo de

retorno `option nat`. Deixamos como exemplo destas operações a função `add_nat` (ver figura 2) que adiciona duas constantes do tipo `nat`.

```
let rec add_nat (n:nat) (m:nat) : nat
  variant { m }
= match m with
  | O -> n
  | S m' -> add_nat (S n) m'
  end
```

Figura 2. Função de adição de duas constantes do tipo `nat` definida em Why3.

O tipo `string` é representado por uma sequência de caracteres, ao invés de um array de caracteres, por se tratar de uma constante, e desta forma mantemos o tipo puro. Dado que um byte é um conjunto de 8 bits, optamos por usar o tipo BV8 (acrónimo para Bit Vector) que são vetores de 8 bits. Todos as estruturas de dados como listas, conjuntos ou mapas, são imutáveis em Michelson e essa propriedade também se reflete em Why3.

Como em todas as linguagens de programação, em Michelson é possível fazer comparações entre constantes. Os tipos comparáveis encontram-se na figura 3.

```
type comparable =
    Int int
  | Nat Natural.nat
  | String (seq char)
  | Bytes (seq Bytes.t)
  | Mutez int
  | Bool bool
  | Key_hash (seq char)
  | Timestamp (seq char)
```

Figura 3. Tipo `comparable` definido em Why3.

Alguns dos tipos na figura 3 podem não ser de compreensão simples e trivial, e desta forma passamos então à sua explicação. O tipo `Mutez` representa *micro-tez*, isto é, a unidade mínima do *token* da *blockchain* Tezos, onde o inteiro representa o número de `mutez` presentes em determinado contrato. Todas as operações envolvendo `Mutez` possuem restrições para evitar a criação de um número negativo de *tokens* ou da sua mistura com outro tipo numérico, e são obrigatoriamente inspecionados os resultados por forma a impedir *overflow/underflow* dos mesmos. Relativamente ao tipo `Key_hash`, este representa o *hash* de uma chave pública. Por fim o tipo *Timestamp* representa uma data que pode

ser escrita em formato legível de acordo com o RFC3339 [10], ou em formato otimizado, como sendo o número de segundos desde o *Epoch*.

De acordo com a especificação em [13] as funções de comparação em Michelson para duas constantes K_1 e K_2 devem devolver um valor inteiro conforme especificado na seguinte equação 1.

$$compare\ K_1\ K_2 = \begin{cases} -1 & \text{se } K_1 < K_2 \\ 0 & \text{se } K_1 = K_2 \\ 1 & \text{se } K_1 > K_2 \end{cases} \quad (1)$$

De maneira a cumprir a especificação foi necessário implementarmos as nossas próprias versões para as referidas funções de comparação. A título de exemplo a figura 4 ilustra a implementação da função de comparação para o tipo `bool`.

```
let compare_bool (a b: bool) : int =
  match a, b with
  | False,True -> (-1)
  | True,False -> 1
  | _,_ -> 0
  end
```

Figura 4. Função de comparação do tipo `bool`.

A pilha de execução da linguagem Michelson é constituída por dados ou instruções, como tal definimos o tipo `data` como mostra a figura 5.

```
type data =
    Instruction instruction
  | Comparable comparable
  | Key data
  | Unit
  | Some data
  | None
  | List (list data)
  | Pair (data , data)
  | Left data
  | Right data
  | Set (set comparable)
  | Map (map comparable data)
  | Big_map (map comparable data)
 with instruction =
  | Seq_i (instruction , instruction)
  ...
```

Figura 5. Definição do tipo `data` em Why3.

É relevante referir que escolhemos uma lista imutável para representação em Why3 da pilha de execução da linguagem Michelson, sendo então o tipo `stack_t` definido da seguinte forma: `type stack_t = list data`.

3.2 Função Eval

Nesta subsecção apresentamos ao leitor alguns detalhes da função `eval`, que a qualquer programa Michelson atribui o seu valor semântico. Por outras palavras, esta função é parte integrante da semântica formal, e como tal queremos que ela além de ser uma função matemática, seja também uma função lógica, e para este tipo de funções necessitamos de fornecer ao Why3 alguma medida de terminação, denominado variante. Mais ainda, demonstramos que é uma função total garantindo a sua terminação através da anotação da mesma com o variante que decresce ao longo de cada execução.

Tendo em conta que Michelson é uma linguagem com uma tipagem estática forte, qualquer programa bem tipado escrito nesta linguagem só admite 3 tipos de *runtime errors*, designadamente, divisão por zero, exaustão de *token*, ou exaustão de gás. No caso da divisão por zero, o problema é resolvido recorrendo ao tipo opcional `Option`, ao qual se procede ao respetivo *pattern matching* garantido assim que erros deste tipo não irão terminar abruptamente a execução do programa quando o divisor for zero, sendo então o resultado da divisão `None`. Os restantes dois casos são os que se referem a algo que não podemos prever. Por exemplo, se um utilizador não providenciar *tokens* (*i.e.* tez) suficientes para um determinado contrato executar (*e.g.* fundos insuficientes para uma transação) o programa irá terminar e estamos perante o caso de exaustão de *token*. Para este caso, definimos a exceção `Abort stack_t`. Finalmente, o último tipo de *runtime error* que podemos obter é a exaustão de gás. Sendo que um *smart contract* não é nada mais que um programa, este tem um custo computacional. Este custo é denominado no protocolo Tezos como sendo *gas*, que a determinada instrução associa um valor computacional. À data de escrita deste artigo, não conhecemos nenhum modelo de custo (*i.e.* gás) formal que permita calcular quanto gás irá gastar determinada transação ou execução de contrato. Voltaremos a abordar este tema na secção 4. Sabemos então que um programa Michelson pode terminar abruptamente devido à exaustão de gás, e para tal definimos a exceção `Out_of_fuel`.

A figura 6 contém parte da função `eval`, onde podemos observar que a mesma recebe como input uma pilha (`inStack`), a quantidade de gás que dispõe naquele momento (`fuel`) e uma instrução (`instr`). Observamos ainda que contém 3 anotações, sendo duas delas referentes às exceções que pode lançar, e a terceira sobre o variante, que permite com recurso a *provers* externos ao Why3, verificar a sua terminação.

Apresentamos agora na figura 7 o processo de avaliação da instrução `EDIV` pela função `eval`. Esta instrução executa a divisão euclidiana entre constantes do tipo inteiro e natural, sendo que o seu resultado é um par contendo o quociente e o resto. Visto que temos dois tipos de constantes numéricas, existem quatro possibilidades para os tipos dos parâmetros desta instrução, designadamente `int`

```
1   let rec eval inStack fuel instr
2     raises  { Abort }
3     raises  { Out_of_fuel }
4     variant { fuel }
5   = if fuel = 0 then raise Out_of_fuel
6     else match instr with
7       (* Control Structures*)
8       | FAILWITH _ -> raise (Abort inStack)
9       | NOOP -> inStack
10      | Seq_i (a,b) -> eval (eval inStack (fuel-1) a) (fuel-1) b
11      | IF_NONE (bt, bf) -> match eval_data (peek inStack) with
12            Comparable (Bool b) ->
13            if b then eval inStack (fuel-1) bt
14            else eval inStack fuel bf
15          | _ -> raise (Abort inStack)
16          end
17      end
```

Figura 6. Parte da função `eval` definida em Why3.

→ `int` (linha 3), `int` → `nat` (linha 6), `nat` → `int` (linha 9) e `nat` → `nat` (linha 12). No entanto podemos observar que só existe aqui um tipo retorno, `option (pair int nat)` o tipo `option` para salvaguardar que o programa não termina devido a uma divisão por zero, e o par constituído pelo quociente (um número inteiro) e o resto (número natural).

3.3 Interpretador OCaml Correto por Construção

A plataforma Why3 oferece ao utilizador um mecanismo de extração de código executável, a partir de uma implementação WhyML verificada. Atualmente, é possível extrair implementações corretas por construção em OCaml, C e CakeML. Nesta secção apresentamos uma implementação OCaml do interpretador de Michelson, obtido automaticamente a partir da implementação WhyML da função `eval`.

Quando comparada com o esforço de especificação e prova desenvolvido, a interação com o mecanismo de extração do Why3 é relativamente simples. No nosso caso concreto, e assumindo que a implementação WhyML do interpretador de Michelson está contida no ficheiro `michelson_eval.mlw`, utilizamos a seguinte linha de comandos:

```
> why3 extract -D ocaml64 michelson_eval.mlw \
    -o michelson_eval_ocaml.ml
```

O mecanismo de extração encontra-se disponível como um sub-programa da plataforma Why3, com a abreviatura `why3 extract` ou `why3_extract`. A opção `-D` constitui-se como um dos aspetos centrais da maquinaria de extração. Através desta opção especificamos um *driver* que deve guiar todo o processo de extração.

```
| EDIV -> let x,t1 = pop inStack in
    let y,t2 = pop t1 in
    (match eval_data x, eval_data y with
    | Comparable(Int x), Comparable(Int y) ->
        if y = 0 then None
        else let res,rem = div x y,mod x y in
           Some (Pair(Comparable (Int res), Comparable(Nat (to_nat rem))))
    | Comparable(Int x), Comparable(Nat y) ->
        if is_zero_nat y then None
        else let res,rem = div x (eval_nat y),mod x (eval_nat y) in
          Some (Pair(Comparable (Int res), Comparable(Nat (to_nat rem))))
    | Comparable(Nat x), Comparable(Int y) ->
        if y = 0 then None
        else let res,rem = div (eval_nat x) y, mod (eval_nat x) y in
          Some (Pair(Comparable (Int res), Comparable(Nat (to_nat rem))))
    | Comparable(Nat x), Comparable(Nat y) ->
        if is_zero_nat y then None
        else let res,rem = div_nat x y, modulo_nat x y in
          Some (Pair(Comparable (Nat res), Comparable(Nat rem)))
    | _ -> raise (Abort inStack) end) :: t2
```

Figura 7. Avaliação da instrução `EDIV` pela função `eval`.

O propósito de um *driver* é dividido em dois aspetos essenciais: definir um *printer* de código para a linguagem-alvo; definir uma substituição entre construções da linguagem WhyML e expressões da linguagem-alvo. No nosso caso, utilizamos o *driver* `ocaml64` visto que é nosso propósito gerar código OCaml para uma arquitetura de 64-bits. Este *driver* define um conjunto de substituições entre símbolos da biblioteca *standard* WhyML e expressões OCaml, como ilustra o seguinte excerto:

```
module int.Int
  syntax val zero "Z.zero"
  syntax val one  "Z.one"

  syntax val (=)  "Z.equal %1 %2"
  ...
end
```

Como podemos observar, o tipo *int* da linguagem WhyML é traduzido para o tipo dos inteiros de precisão arbitrária da biblioteca *zarith*. As constantes 0 e 1 (linguagem WhyML), assim como a função de comparação entre dois inteiros, são traduzidas para construções equivalentes desta biblioteca.

O programa OCaml obtido apresenta uma estrutura em tudo semelhante à que observamos para o programa WhyML. De facto, ambas as linguagens apresentam vários pontos em comum, nomeadamente ao nível da organização do código. A função `eval` obtida apresenta a seguinte estrutura:

```
exception Out_of_fuel

let rec eval (inStack: data list) (fuel: Z.t) (instr: instruction) :
  data list
= if Z.equal fuel Z.zero then raise Out_of_fuel
  else ...
```

Notamos que a comparação entre dois inteiros, realizada no código WhyML original, é agora transformada numa comparação entre dois inteiros de precisão arbitrária. O restante da função `eval` apresenta uma estrutura praticamente homomórfica à do código WhyML, a não ser pelas pequenas diferenças sintáticas entre a linguagem de programação do Why3 e o OCaml.

Uma descrição detalhada do funcionamento e maquinaria subjacente ao mecanismo de extração do Why3 pode ser encontrada na tese de doutoramento do segundo autor [12]. Este trabalho contém, além de diversos exemplos, uma prova de correção do mecanismo de extração implementada com base numa semântica formal *big-step* e num sistema de tipos com efeitos.

4 Conclusões e Trabalho Futuro

Neste artigo apresentámos um interpretador Michelson como primeiro esforço para a formalização de uma plataforma de verificação de *smart contracts* para a *blockchain* Tezos. Fizemos ainda um paralelismo entre uma versão executável em OCaml extraída da plataforma de verificação Why3 e a sua especificação na referida plataforma. Com todas as aplicações que se tornaram possíveis de implementar com o aparecimento das tecnologias de *blockchain*, sentimo-nos motivados no sentido de continuar este esforço de verificação formal para que todos possamos usufruir delas de forma "segura".

Como trabalho futuro podemos então estabelecer duas metas. A mais imediata e na qual estamos já a trabalhar consiste em escrever uma relação de tipagem para a linguagem Michelson. Esta possibilitará introduzir um conjunto de anotações na função `eval` e posteriormente mostrar que estes são na verdade pontos inacessíveis no interpretador. Como objetivos mais ambiciosos, pretendemos fazer uma análise e consequente modelo de custo para o `gas`. Queremos ainda testar a versão executável extraída da plataforma Why3 como possível substituta para a versão atual da plataforma Tezos baseada em GADTs.

Referências

1. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06). pp. 3–12. Ieee (2006)
2. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming

Languages and Analysis for Security. pp. 91–96. PLAS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2993611, http://doi.acm.org/10.1145/2993600.2993611
3. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
4. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European Symposium on Programming. pp. 125–128. Springer (2013)
5. Foroglou, G., Tsilidou, A.L.: Further applications of the blockchain. In: 12th Student Conference on Managerial Science and Technology (2015)
6. Goodman, L.: Tezos: A self-amending crypto-ledger position paper (2014)
7. Matteson, S.: Software failure caused $1.7 trillion in financial losses in 2017 (2018), [Online; https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017 accessed 3-July-2019]
8. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008) (2008)
9. Nehai, Z., Bobot, F.: Deductive proof of ethereum smart contracts using why3. arXiv preprint arXiv:1904.11281 (2019)
10. Newman, C., Klyne, G.: Date and Time on the Internet: Timestamps. RFC 3339 (Jul 2002). https://doi.org/10.17487/RFC3339, https://rfc-editor.org/rfc/rfc3339.txt
11. Nomadic Labs: A specification of michelson in coq to prove properties about smart contracts in tezos (2019), [Online; https://gitlab.com/nomadic-labs/mi-cho-coq accessed 28-May-2019]
12. Parreira Pereira, M.J.: Tools and Techniques for the Verification of Modular Stateful Code. Theses, Université Paris-Saclay (Dec 2018), https://tel.archives-ouvertes.fr/tel-01980343
13. Tezos Foundation: Michelson: the language of smart contracts in tezos (2019), [Online; https://tezos.gitlab.io/master/whitedoc/michelson.html#semantics accessed 15-April-2019]
14. Yang, Z., Lei, H.: Fether: An extensible definitional interpreter for smart-contract verifications in coq. IEEE Access **7**, 37770–37791 (2018)