Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Análisis de Contratos Inteligentes Usando Cláusulas de Horn

Autor: Víctor Pérez Carrasco
Tutor(a): Manuel Hermenegildo Salinas

Madrid, 16 de agosto de 2020

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Análisis de Contratos Inteligentes Usando Cláusulas de Horn

16 de agosto de 2020

Autor: Víctor Pérez Carrasco
Tutor: Manuel Hermenegildo Salinas
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En software crítico, es común la necesidad de demostrar que las aplicaciones se atienen a unas especificaciones sobre el consumo de recursos. Para ello, el análisis estático se postula como una solución claramente superior al testing, ya que este último se limita a probar la no conformancia mediante la búsqueda de un caso que no cumpla con las condiciones, mientras que solo el primero puede asegurar que las especificaciones se respetan para un conjunto infinito de posibles entradas. En este trabajo, nos centramos en el análisis de un tipo de software crítico cuya popularidad ha crecido considerablemente en los últimos años: los contratos inteligentes.

La propia naturaleza de los contratos inteligentes y de las blockchain hace que el análisis del consumo de recursos en estos sea de gran importancia. El alto factor de replicación de estos programas en los nodos de las plataformas distribuidas hace que hasta la operación más sencilla vea su coste computacional multiplicado y que el espacio de almacenamiento requerido por estos crezca vertiginosamente. Esta situación, unida a la mutabilidad y variabilidad de las plataformas en cuanto a lenguajes utilizados y modelos de coste empleados, empuja a pensar que una aproximación *genérica* y *configurable* a este problema sea una opción interesante. Esto contrasta con las soluciones propuestas anteriormente que son más específicas para un lenguaje de contratos o una plataforma dados.

Siguiendo esta línea, en este trabajo presentamos un enfoque flexible y viable al análisis estático de contratos inteligentes, particularizado a la plataforma Tezos. Para ello implementamos un traductor de Michelson a Cláusulas de Horn, las cuales se analizan con la herramienta `CiaoPP`. El análisis realizado se enfoca principalmente en estimar el consumo de *gas*, un recurso virtual que refleja el tiempo de ejecución de un programa. Dicho análisis se apoya en un modelo de coste que hemos expresado en un lenguaje de aserciones, y que recoge la información en términos de consumo de recursos de las distintas instrucciones presentes en el lenguaje Michelson. De este modo, se presenta la posibilidad de realizar un análisis de recursos altamente configurable que se adapte a las modificaciones que pueda sufrir la plataforma a estudiar a lo largo del tiempo.

Presentamos resultados experimentales del análisis estático de contratos inteligentes obtenidos con la técnica propuesta que muestran que la solución es factible, y puede ser precisa y eficiente. Estos resultados son alentadores, y sugieren que esta técnica es una vía prometedora para continuar explorando en el futuro.

Abstract

In critical software, it is usually needed to ensure the conformance of applications with respect to specifications that constrain resource usage. In order to achieve this, static analysis stands as a clearly superior solution compared to testing, as the latter is only capable of proving that a program does not meet the necessary conditions by searching for a non-compliant case, whereas only the former can assure that specifications are complied with for an infinite set of possible inputs. In this work, we will focus on the analysis of a kind of critical software which is becoming increasingly popular in the last few years: smart contracts.

The very nature of smart contracts and blockchain makes resource analysis on these of great importance. The high replication factor presented by these programs in distributed platform nodes leads to the multiplication of the computational cost of running even the simplest operation and the rapid growing of the consumed storage space. This situation, combined with the mutability and variability that these platforms present in terms of languages and cost models, compels one to think that a *generic* and *configurable* approximation to this problem is an interesting option. This is in contrast with previous approaches that are more specific for a contract language or platform.

Following this line, in this work we present a flexible approach to the static analysis of smart contracts, using the Tezos platform as the concrete case of application. To this end, we have implemented a Michelson to Horn Clause translator and run `CiaoPP`'s static resource analysis on Tezos smart contracts represented this way. The analysis focuses on analyzing *gas* consumption, a virtual resource which reflects a program's execution time. Such analysis relies on a cost model that we have expressed using an assertion language, which represents the resource consumption for each Michelson instruction. This way, it is possible to perform a highly configurable resource analysis, capable of adapting to possible changes that the platform might suffer over time.

We present experimental results on the static analysis of smart contracts obtained using the proposed approach which show that the approach is feasible, and can be accurate and efficient. These results are encouraging, and suggest that the approach is indeed a promising avenue for future research.

Contents

Chapter 1

Introduction

Since its inception in 2008, blockchain technology [1] has attracted companies from a huge array of sectors, such as banking, retail, or even gaming. These companies use smart contract platforms, such as Ethereum [2] or Tezos [3], which allow performing secure transactions without the need of third parties.

The basic idea behind smart contracts [4] is that it is possible to implement a system which formalizes and secures digital relationships on current software and hardware. This system must be designed in a way that discourages breachers from trying to perform an attack, by making it prohibitively expensive for them.

Being such critical components, languages used to write smart contracts should not only be robust against any class of attack, but also provide a user interface to communicate transaction semantics. As a result, these languages need to be secure, avoiding unwanted behaviour and allowing the programmer to write a program which performs exactly as desired, while allowing other parties to read and understand its behaviour.

In the case of the Ethereum platform, its main focus is to provide programmers with a Turing-complete programming language to write decentralized applications, which can range from pure financial applications to complex cloud computing environments. In order to do so, Ethereum provides developers with two high level programming languages which compile to Ethereum Virtual Machine (EVM) bytecode. The most popular choice in this regard is Solidity, a C-like language, which has established itself as the dominant language to write smart contracts.

On the other hand, the focus of the Tezos platform is safety, and an important enabler in this regard is the VM language used, Michelson. Unlike the aforementioned languages used in Ethereum, Michelson was explicitly designed to facilitate formal verification. In other words, the developers of Tezos smart contracts should be able to prove that their contracts will behave exactly as desired before running them. Because of this design approach, Michelson does not include features such as polymorphism or named functions, as, unlike the languages used in the Ethereum platform, Michelson does not intend to be a general purpose language, as its only raison d'être is to implement pieces of business logic.

1.1. The Tezos Michelson contract language

As we have stated above, smart contract languages should provide a user interface to communicate the semantics of transactions. Whereas Ethereum's approach to this goal is to provide readable languages which compile to assembly-like EVM bytecode, Tezos' Michelson was designed to be a readable compilation target. This way, developers can opt for writing contracts in a higher-level language, such as Python or Haskell, and compile the result to Michelson, as an alternative or in addition to using Michelson directly. Following on from the intention to provide a language designed with formal verification in mind, this simplicity allows developers to easily implement their own tools to verify properties of their code or even do it by hand.

As we can see in Listing 1.1, these programs are divided into three distinct parts. The first two sections comprise the contract's parameter and storage types declaration. A parameter is the value used when calling a contract and the storage is the memory of the contract. A call to a contract may or may not update its storage, as we will see when we dive into a contract's main section: its code.

The code section carries a contract's semantics, its purpose, so it should be clear and precise. Michelson is run by an interpreter which we can conceive as a pure function: it receives an input stack and returns a result stack, without altering its environment. This input stack will only contain a pair consisting of the calling parameter and the contract's storage at that moment. Regarding the result stack, it will only contain a pair consisting of a list of internal operations to be executed when the contract returns and the resulting contract storage, which may be the result of an alteration to the initial storage, the same unaltered value or a completely different one. These internal operations which the contract may return can be of three kinds: transactions, to transfer tokens to an account or make a call to another contract, providing the parameter to use (two operations that, as we will see in a following paragraph, are almost equivalent); originations, to create a new smart contract; or delegations, to assign tokens to another account without transferring them, as the recipient will not be able to spend them and the sender may take them back at any moment.

Just as the Michelson interpreter, Michelson instructions are also pure functions, as they simply extract values from the top of the input stack and push values into the resulting stack. Thus, the Michelson language replaces variables with stack elements. This simplifies the language but in return makes writing Michelson contracts somewhat harder. We will take the following instructions from the smart contract above to illustrate this property of Michelson contracts: `CAR`, `PUSH int 1` and `ADD`. The first instruction extracts a pair from the top of the stack and inserts its first element. Then, the second instruction pushes a constant integer of value 1. Finally, the last instruction extracts two integers from the top of the stack and pushes the result of summing both of them. The mathematical definition of these instructions as pure functions can be seen in Equation 1.1, Equation 1.2 and Equation 1.3 respectively:

$$CAR \;/\; (Pair\, a\, \_) \;:\; S \;=>\; a \;:\; S \tag{1.1}$$

$$PUSH\; int\; 1 \;/\; a \;:\; S \;=>\; 1 \;:\; a \;:\; S \tag{1.2}$$

$$ADD \;/\; 1 \;:\; a \;:\; S \;=>\; (1 + a) \;:\; S \tag{1.3}$$

```
parameter (option string) ;
storage (pair (pair nat nat) (map nat string)) ;
code { DUP ;
       CAR ;
       IF_NONE
         { CDR ;
           DUP ;
           CAR ;
           DIP { CDR ; DUP } ;
           DUP ;
           CAR ;
           SWAP ;
           DIP { GET } ;  # Check if an element is available.
           SWAP ;
       # Put NONE on stack and finish.
           IF_NONE
             { NIL operation ; PAPAIR }
         # remove the entry from the map.
             { DROP ;
               DUP ;
               DIP { CAR ; DIP { NONE string } ; UPDATE } ;
               DUP ;
               CAR ;
               PUSH nat 1 ;
               ADD ;
               DIP { CDR } ;
               PAIR ;
               PAIR ;
               NIL operation ;
               PAIR } }
     # Arrange the stack.
         { DIP { DUP ; CDAR ; DIP { CDDR } ; DUP } ;
           SWAP ;
           CAR ;
       # Add the element to the map and increment the second number.
           DIP { SOME ; SWAP ; CDR ; DUP ; DIP { UPDATE } ; PUSH
              nat 1 ; ADD } ;
       # Cleanup and finish.
           PAIR ;
           PAIR ;
           NIL operation ;
           PAIR } }
```

Listing 1.1: A Michelson contract example.

In addition to this fact, the type of a stack, i.e., its length and the type of the elements it contains, can be known beforehand at any program point by performing a trivial static analysis, due to the aforementioned limitations imposed on the Michelson language and the deterministic semantics of its instructions. This way, Michelson assures that the execution of a contract will only fail if intended to do so (the `FAILWITH`

instruction is called), not enough tokens are provided, or due to *gas* exhaustion, a core topic of this writing which we will cover more in detail.

Despite its similarities to assembly, Michelson includes some high-level data structures, such as lists or maps, which can be accessed and modified at execution time, stored by contracts and, in short, will be treated exactly as primitive types.

These Tezos smart contracts reside inside blocks in the Tezos blockchain together with their private data storage. This way, in order to execute a Tezos smart contract, the user has to perform a transaction carrying data, the parameter, to its associated account. This process can be seen as a remote procedure call (RPC) which updates the smart contract storage, which provides read-only access to users otherwise.

1.2. Resource consumption in Michelson

Due to the nature of blockchain platforms, smart contracts will be stored in every single node running the chain, so its storage will be replicated in all of them and any call to a smart contract will be executed on every node. This fact lead Tezos to include an upper bound in execution time and storage, as well as a fee associated with running a contract or increasing its storage size, establishing a cost per allocated byte to restrain storage use.

In order to limit execution time, the Tezos platform makes use of a concept called "gas". Being an interpreted language, every Michelson instruction has an associated cost in *gas*, which is accumulated by the Michelson interpreter. This way, if a transaction exceeds its allowed *gas* consumption, its execution is stopped and its effects, reverted. However, even if a transaction does not succeed because of *gas* exhaustion, it is included in the blockchain and the fees are taken.

The aforementioned properties, *gas* and *storage size*, can be seen as resources consumed by the execution of a smart contract. On the one hand storage size is a physical resource, as it can be measured just by inspecting a smart contract. On the other hand, *gas* is a virtual resource that reflects the cost in execution time (a physical resource). By regarding these properties as resources, we can associate a cost to running a contract. A cost which will be expressed in terms of *gas* consumed or "burned" and allocated bytes.

With that in mind, knowing the cost of running a contract beforehand can be beneficial for users, as it would allow them to know how much they would be charged for the transaction and if *gas* limits would be exceeded. Tezos provides users with an ad-hoc solution in order to perform such an a-priori estimation of the resource consumption of a contract: it allows users to dry run a smart contract in their own node before making a transaction. This way, a user can know the cost of running a contract for a specific input value.

This way of testing the cost of running a transaction allows the user to make sure that a transaction will succeed for a specific pair of $(parameter, storage)$ values, but it cannot assure that a change in one of these two values will not result in undesired behaviour, which could lead to excessive execution costs. If the possible pairs of input values (or some parts of them) are not known a priori, and an infinite of very large set

of values is possible, testing for all those instances can quickly become inefficient or simply infeasible.

In such cases a formal verification-based approach is preferable. At the same time, as mentioned at the beginning, being able to verify properties is one of the strong motivations and fundamental tenets of the Tezos approach. It does make sense thus to focus on formal analysis and verification, not only of classic correctness properties, but also of the resource-oriented aspects of the platform.

1.3. Towards formal verification of contract resources

As mentioned above, the importance of formal verification for critical software such as smart contracts resides in the fact that it can be used to *prove* a certain property of a piece of code, such as the absence of bugs or the cost of running a transaction.

As a result, formal verification of smart contracts, and in particular analysis and verification of their resource consumption is an increasingly popular topic. It is however also a challenging one. At the same time, there are now many different platforms, using different smart contract languages and cost models. These models often take into account different resources and count them in a different, platform-specific way. Furthermore, within each platform, the models can also evolve over time. As a consequence, the few existing resource analysis tools for smart contracts, such as GASTAP [5], GASOL [6], and MadMax [7] (see Sec. 1.5), tend to be quite specific, focusing on just a single platform or language, or on small variations within a set.[1]

In order to address the challenge posed by the rapid development of smart contracts platforms, we propose a different approach based on *developing instead analysis tools that are generic and can be flexibly configured to adapt to changes*, both within platforms and from platform to platform. Our objective in this work is to develop an example of such a tool.

To this end, we will use as a fundamental tool the `CiaoPP` framework [8]. This framework allows the analysis, transformation, and verification of programs written in different languages and for different platforms by transforming such source programs and the associated cost models into an intermediate representation, based on Horn clauses [9]. This approach has been proven to allow the analysis and verification of many properties, including *resource consumption*: different resources and cost models are user-definable [10], by using the `Ciao` assertion language [11]. From these models, `CiaoPP` can perform a class of analyses on the input code, and in particular, it includes a parametric, interval-based resource analysis of the input program based on these user-defined cost models (see [12] and its references). This approach builds on initial work for the inference of upper bounds on task granularity in automatic program parallelization [13, 14], which evolved to deal with other types of approximations (e.g., lower bounds [15]), user-defined resources [10], integrated multi-variant resource analyses (i.e., path and context sensitive resource analyses [16]), and analysis of properties such as time or energy [17, 18, 12]).

One of our objectives in this work is to show that the generic approach of `CiaoPP` can be used to analyze smart contracts from different platforms written in different

[1] We will return to discuss this and other relevant related work in Section 1.5.

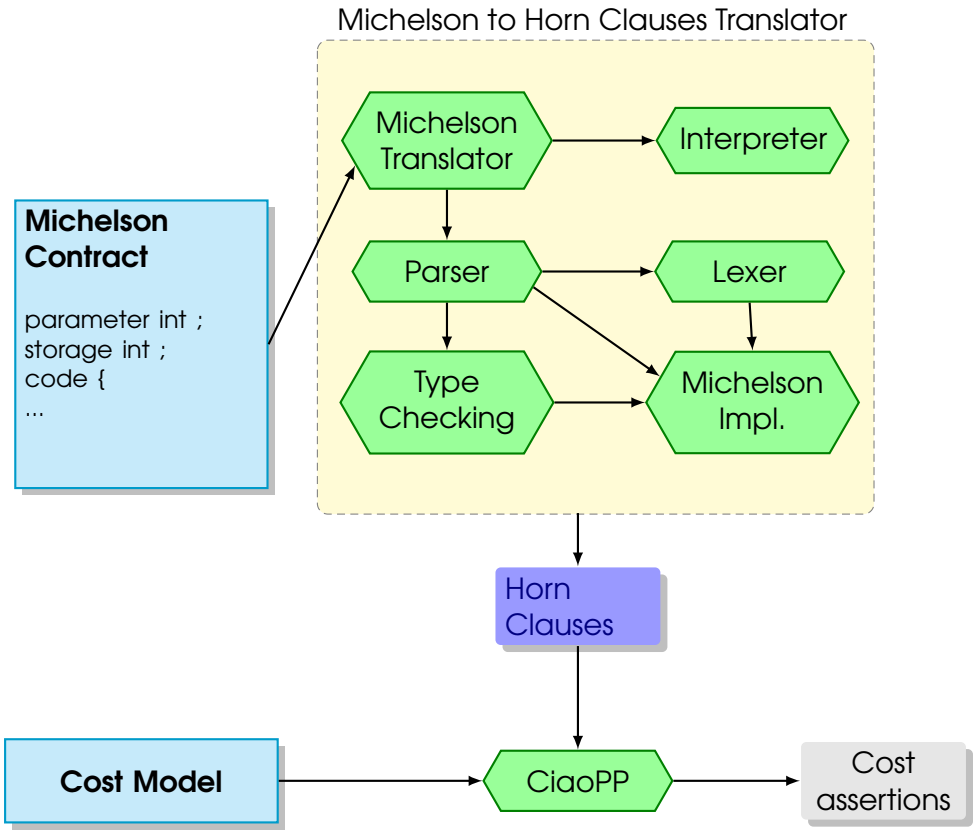

Figure 1.1: Michelson contract analysis process.

languages, i.e., that *it is not necessary to develop specific analyses for smart contracts and a generic tool such as* `CiaoPP` *can be used for this purpose*. However, *specific transformations need to be developed* to capture the semantics of smart contracts, as well as *cost models* in order to capture the gas model. Figure 1.1 summarizes this generic approach applied to Michelson smart contracts.

1.4. Plan and outline

As mentioned before, our objective is to prove by construction that it is possible to perform cost analysis on smart contracts from different platforms written in different languages with a common set of tools, through the two-pronged approach of defining resources via assertions and translating to an intermediate language. In order to do so, we will perform and report on the following tasks:

- Our first step will be to **compile the source code to obtain its Horn clause representation**. This can be easily achieved by coding a simple parser and a translator. It is worth mentioning that the resulting Horn clause program does not need to run, i.e., it will just be used to perform the analysis. Furthermore, this technique can be made even simpler, since, if there are any operations with no impact on the cost analysis, they can be omitted. A characteristic of Michelson is that it is a stack-based language. This means that every instruction receives a stack, pops its input variables from the stack and pushes its

output variables to it, passing the resulting stack to the next instruction. The stack-based nature of Michelson makes a direct translation of its instructions and contracts directly as functions or predicates in the intermediate form less natural. Thus, in order to obtain a translation that can make use of the built-in support for variables, control, and parameter passing in the analyzers the stack will be compiled away in the translation process. This translation process is discussed in Chapter 2.

- After obtaining a compiler from Michelson contracts to Horn clauses, the next big task to tackle is **developing a cost model** and using it with `CiaoPP` to **perform the resource analysis**. We will annotate the cost of each instruction as found in our cost model using `CiaoPP` assertions. By doing so, the framework will make use of our previously-developed cost model to perform a parametric analysis taking into account input data sizes and returning upper- and lower-bound cost functions for the queried resources. We will consider four paths to accomplish this goal:

  - Go through the source code, if available, and look up what operations are performed when executing an operation that consume resources.
  - Perform program slicing on such source code in order to obtain the parts of the code that will be useful for our cost analysis.
  - Run the `CiaoPP` tool on the source code to get a similar result to the previous case.
  - Use an existing cost model provided by the smart contracts platform where we will run the analysis on.

  We will discuss pros and cons of each one of these options.

  Another characteristic of the Michelson language is that is statically typed. This means that we can know the input and output stack types for every instruction at compile time. This will come in hand when developing a cost model, since the translator can discriminate what version of a polymorphic instruction to use in translation time depending on the state of the stack. We will see how in some cases this can be useful, as the cost of an instruction can depend on its input type, but in other cases it will not be necessary to do so.

  In Chapter 3 we will go through the different possible ways to develop a cost model, weighing their pros and cons. This configurable cost model will be used in Chapter 4, which will include some **working examples of the analysis of Michelson contracts**, evaluating its usefulness and correctness.

- Finally, Chapter 5 summarizes our conclusions and comments on ongoing and future work.

1.5. Other related work

In this section we introduce or discuss further some of the related work on static or mixed smart contracts resource analysis and resource analysis in general, that was

not included or mentioned briefly in the previous discussions in the context of the CiaoPP approach.

Regarding GASTAP [5] and its extension GASOL [6], mentioned before, these tools analyze Solidity source code and EVM bytecode to perform different analyses on Ethereum smart contracts, such as upper bounds for *gas* consumption or even potential sources of optimization. In the case of GASOL it also allows users to define their own cost models, so it can conceptually be used to analyze other platforms that execute Solidity smart contracts.

As also mentioned before, another interesting tool used to analyze Ethereum smart contracts is MadMax [7]. A main feature of MadMax is that it analyzes EVM bytecode by using the approach pioneered by CiaoPP of previously translating to an intermediate representation (this is work based on collaborations in previous projects with the CiaoPP team within the ENTRA project [18]). Once translated, it can perform different analyses on the result using an Ethereum-based cost model, such as detecting vulnerabilities that can be easily exploited to block contracts.

As we have seen, these tools have one thing in common: their focus on a single smart contract platform or on a small set of them. In the case of MadMax, it is an interesting tool, but it can only be used to analyze smart contracts on the Ethereum platform. Also, GASOL/GASTAP can be used on Solidity smart contracts, but, again, this leaves out a great number of smart contract platforms not using this language.

In a more general context, using abstract interpretation in verification, debugging, and related tasks has now become well-established. To cite some early work, abstractions were used in the context of algorithmic debugging in [19]. Abstract interpretation has been applied by Bourdoncle [20] to debugging of imperative programs and by Comini et al. to the algorithmic debugging of logic programs [21] (making use of partial specifications in [22]), and by P. Cousot [23] to verification, among others. The CiaoPP framework [24, 25, 8] was the first to offer an approach combining abstraction-based verification, debugging, and run-time checking, using a common assertion language; this approach can be seen as the first one to bridge the advantages of static and dynamic languages and approaches. This approach has recently been applied in a number of systems [26, 27, 28, 29] implementing hybrid typing, gradual typing, etc.

Horn clauses are used in many different applications nowadays as compilation targets or intermediate representations in analysis and verification tools [30, 9, 31, 32, 33, 34, 35].

Chapter 2

Michelson to Horn Clauses Translation

In this chapter, we will cover the implementation process of a Michelson to Horn Clauses translator. Firstly, we will discuss the design choices behind a Michelson lexer in Section 2.1. Then, we will approach Michelson's simple syntax to implement a Michelson parser in Section 2.2 and a type checking module in Section 2.3. In Section 2.4, we will discuss the implementation of a Michelson interpreter in Prolog to better understand this language's semantics. And, finally, we will address the core part of the translator, which transforms the output from the former modules into a working `Ciao` module, in Section 2.5; as well as an overview of the translation process by means of an example in Section 2.6.

It is worth noting that the cost of running the translator is negligible compared to that of analyzing its output. Due to this fact, more effort had to be put into adapting this translator's output to `CiaoPP`'s needs instead of optimizing the translation process.

2.1. Lexer

As whitespace and comments are not relevant Michelson tokens, this lexer will omit them, as portrayed in Figure 2.1. As seen in Figure 2.2, a Michelson token can be of five types: a type definition, a keyword, a constant, an instruction or a symbol, which is to say, a block opener or closer; or a semicolon.

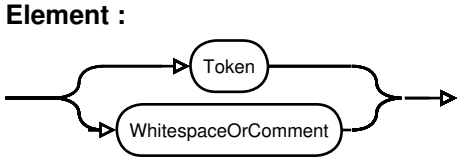

Figure 2.1: Michelson's lexer core behaviour.

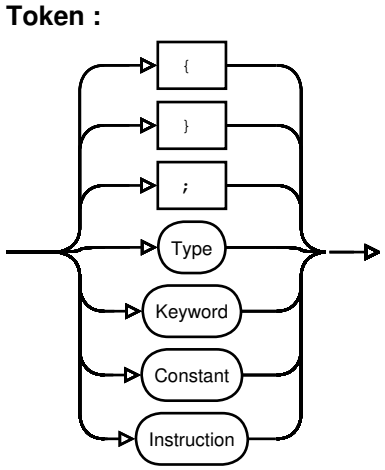

Figure 2.2: Michelson token types.

It is worth mentioning the existence of a sixth token class: annotations. Annotations were left out of the implementation of this tokenizer, as they complicate Michelson syntax, which would result in a more complex parser; while not adding any relevant information in terms of cost analysis nor semantics. Because of this, annotations are treated as comments by our lexer, as shown in Figure 2.3.

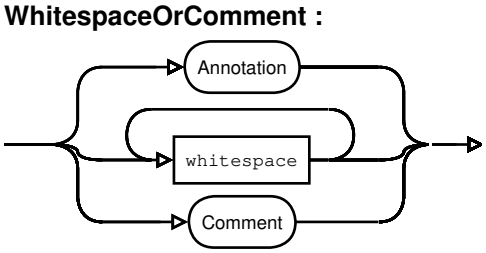

Figure 2.3: Elements omitted by the lexer.

Michelson includes both line and block comments, so both should be contemplated and omitted by the lexer (Figure 2.4).

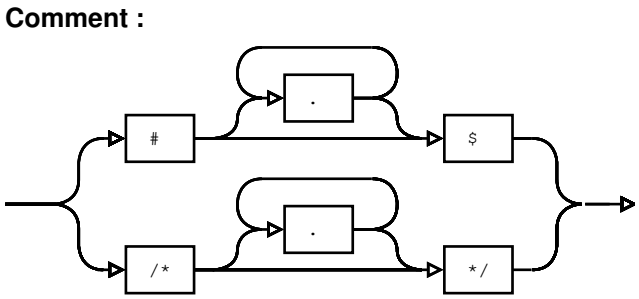

Figure 2.4: Michelson comments treatment.

There exist three types of annotations (Figure 2.5) in Michelson and the three of them are omitted: type annotations, beginning with ":", variable annotations, beginning

with “@” and constructor annotations, beginning with “%".

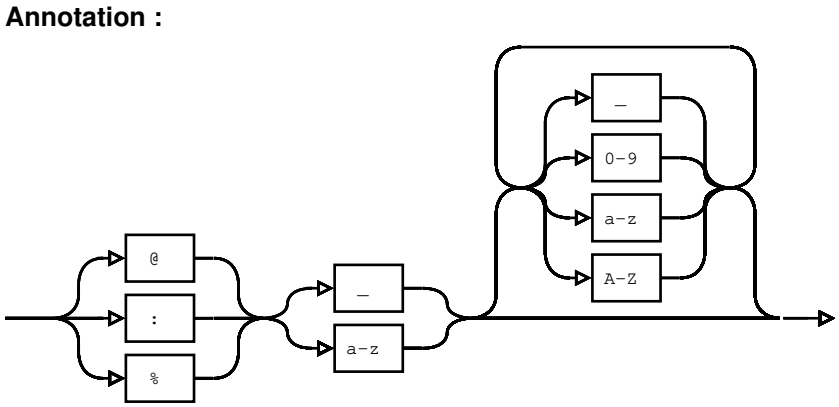

Figure 2.5: Different Michelson annotations.

In Michelson there are eight keywords (Figure 2.6) in addition to type definitions. Keywords “code”, “parameter” and “storage” point out the beginning of each of the sections of a contract; while “Pair”, “Left”, “Right” and “None” and “Some” are used to insert constants. They are used in pairs, union constants, to indicate which of both types is to be used and optional constants, to show if the constant contains a value or not.

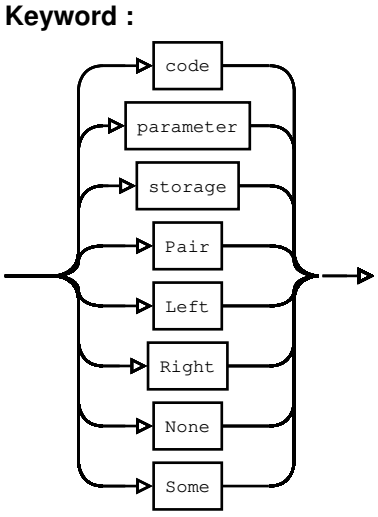

Figure 2.6: Michelson recognized keywords.

Regarding types, we can distinguish two groups: comparable and incomparable types. The main difference between these is the fact that only comparable types are allowed as key types of maps or set elements or can be compared by using `COMPARE` instruction.

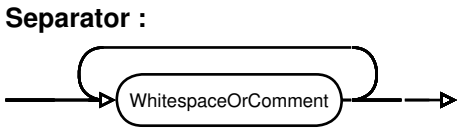

Figure 2.7: Separation between type names.

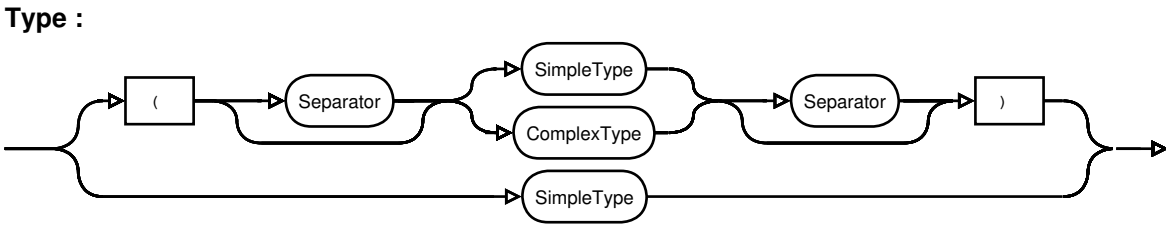

Figure 2.8: Michelson types classification.

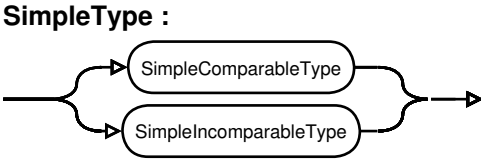

Figure 2.9: Michelson simple types.

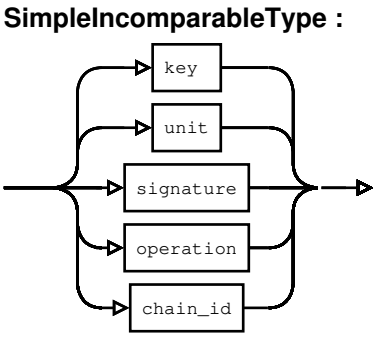

Figure 2.10: Incomparable simple types.

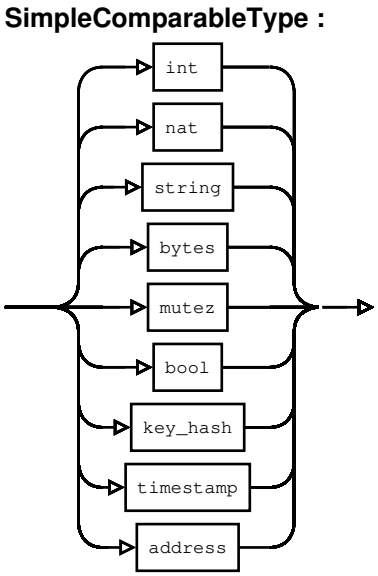

Figure 2.11: Comparable simple types.

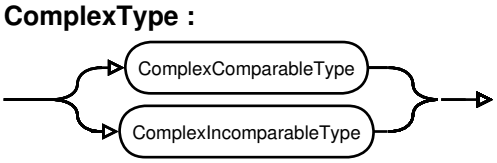

Figure 2.12: Complex types.

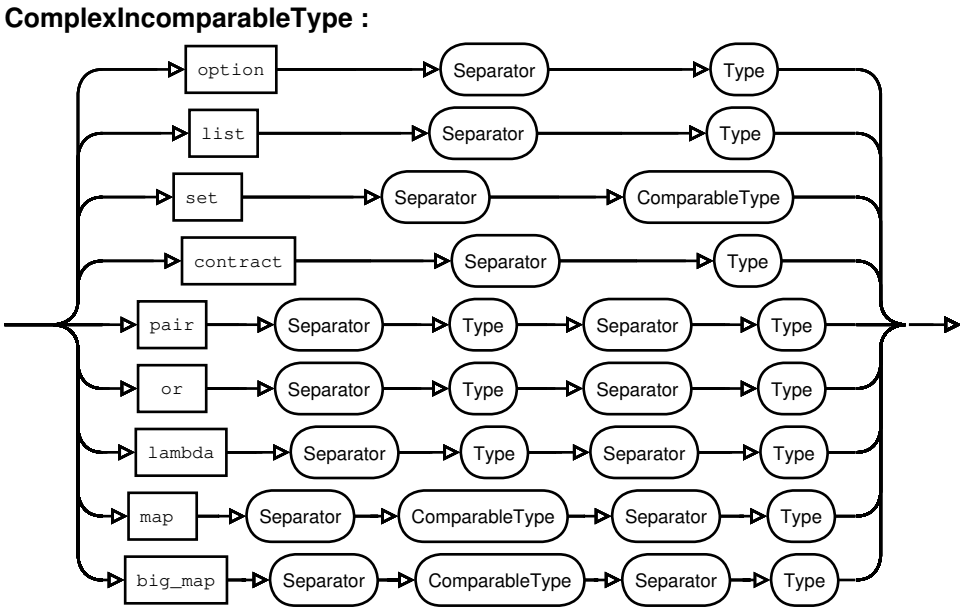

Figure 2.13: Incomparable complex types.

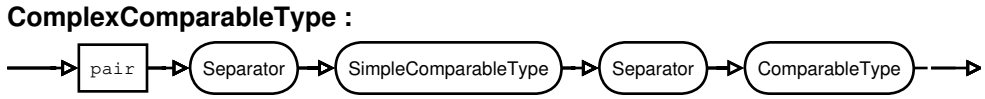

Figure 2.14: Comparable complex types.

There are other restrictions regarding the usage of certain types, but these will be treated by the parser.

As for constants, the lexer recognizes six types of them (Figure 2.15): bytes (or byte sequences), integer numbers, strings, the `Unit` constant and Boolean constants `True` or `False`.

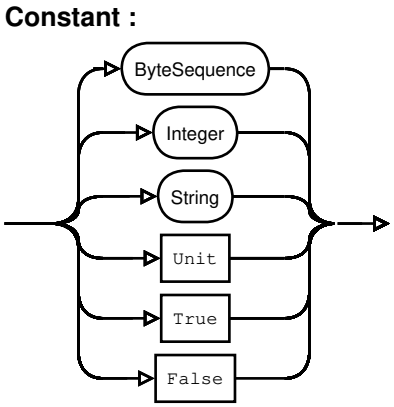

Figure 2.15: Michelson constant types.

Bytes (Figure 2.16) are not stored as a numeric representation of the input, but as a sequence of 8-bit integers. These constants consist of pairs of hexadecimal digits preceded by "0x", which marks the beginning of a bytes constant.

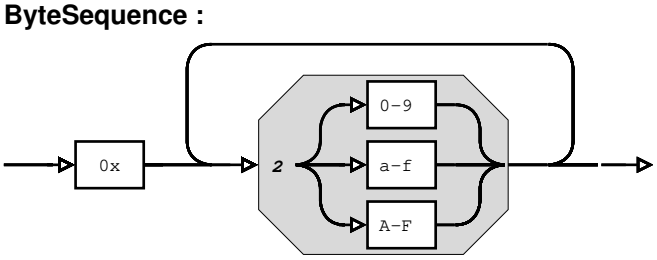

Figure 2.16: Michelson byte sequence constants.

Integer (Figure 2.17) and string constants (Figure 2.18) behave similarly to those found in other languages. Strings also include escape sequences.

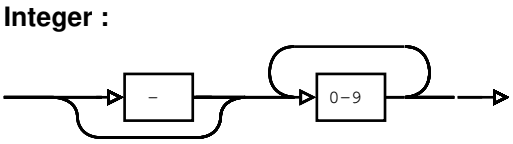

Figure 2.17: Michelson integer constants.

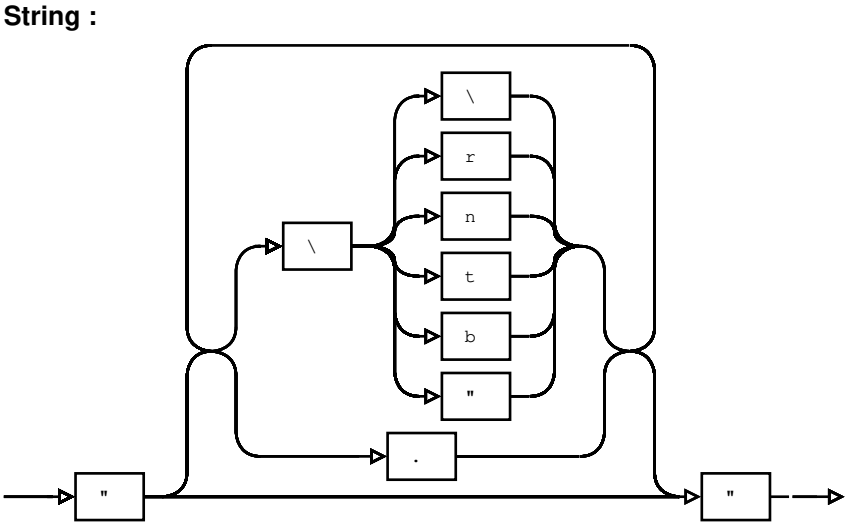

Figure 2.18: Michelson strings.

As far as instructions, there is a vast array of them, which the lexer classifies into nine different categories in order to facilitate the work of the parser, as seen in Figure 2.19. These instruction classes and the parser itself will be treated in greater detail in the next section.

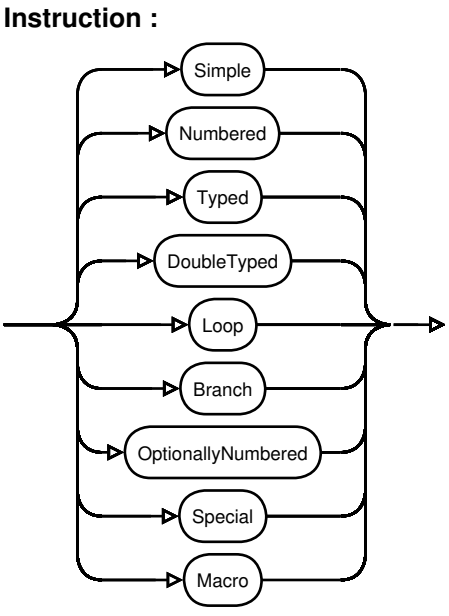

Figure 2.19: Michelson instruction types.

A special type of instructions are DU+P and DI+P instructions. The former instruction extracts the n^{th} element from the stack and inserts it back in its position and in the top of the stack, whereas DI+P extracts n elements from the top of the stack, executes a block of code and pushes these elements back in the stack. Now, the unusual aspect of these instructions is the fact that their numeric parameter can be expressed via the repetition of the letters conforming the instruction itself. This way, by counting the number of "U"s or "I"s in these instructions, we can know the value of the parameter n involved in these operations. Figure 2.20 displays the treatment of these special instructions by the lexer.

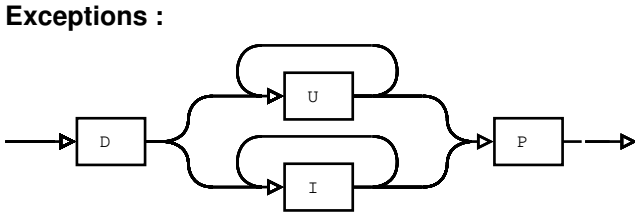

Figure 2.20: Special Michelson instructions.

2.2. Parser

The simplicity of Michelson's syntax can be perceived by examining this Michelson parser.

As mentioned in this thesis introduction, a Michelson contract consists of three parts: parameter and storage type declarations and the code. Each of these sections must be present exactly once in the contract, in any order (Figure 2.21).

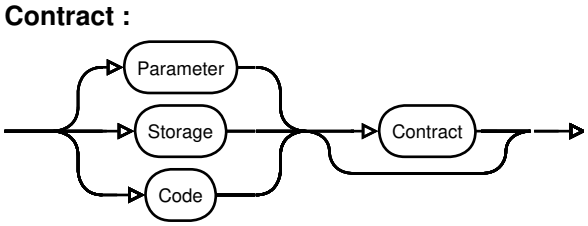

Figure 2.21: Michelson contract syntax.

Parameter (Figure 2.22) and storage (Figure 2.23) sections are very simple, as they just include a keyword and a type, which are the parameter and storage types for the contract, respectively.

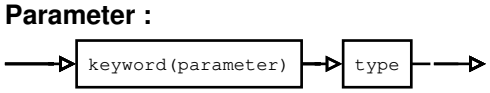

Figure 2.22: Michelson parameter section syntax.

Figure 2.23: Michelson storage section syntax.

It is worth noting that Michelson applies restrictions to the types which can be applied to these fields. Neither storage nor parameter can be of operation type. And, in the case of the storage, neither does Michelson allow contracts to be stored.

The remaining contract section, the code, is where instructions reside. As seen in Figure 2.24, it consists of a sequence of semicolon-separated instructions delimited by a block opener and a block closer.

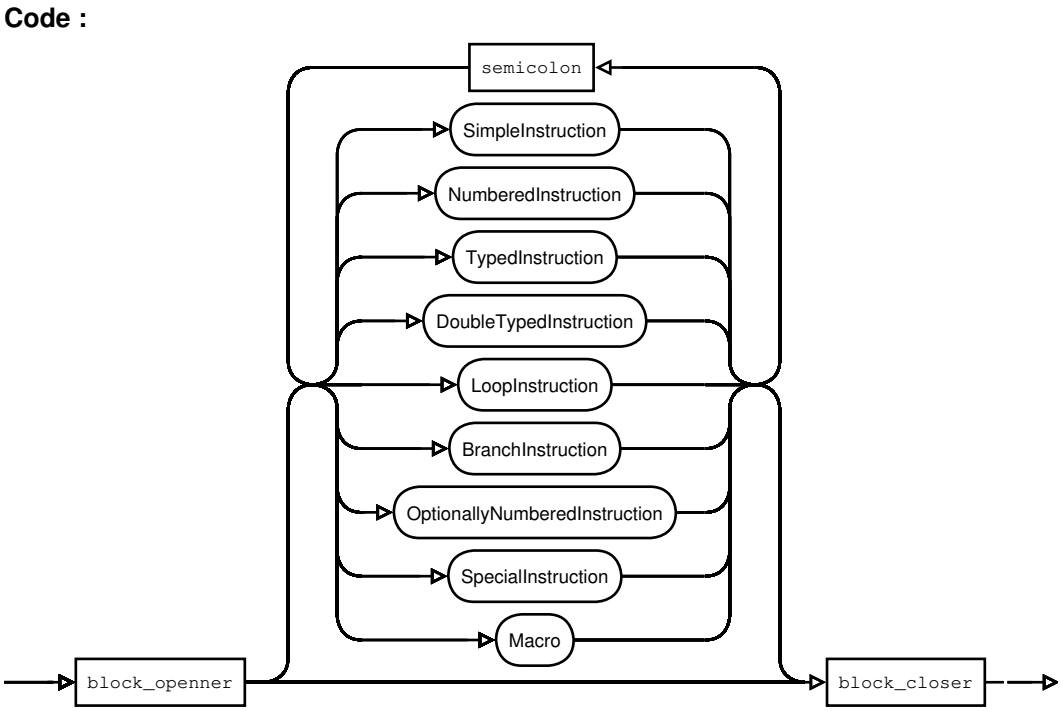

Figure 2.24: Michelson code section syntax.

Now, we will briefly cover each of these instruction classes.

Firstly, simple instructions (Figure 2.25) are instructions that can not receive any additional parameters, such as `ADD` or `PAIR`.

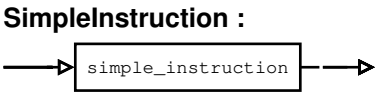

Figure 2.25: Michelson simple instructions syntax.

The following instruction classes are characterized by the fact that they need additional parameters, such as a numeric constant (Figure 2.26) a type (Figure 2.27) or

two of them (Figure 2.28).

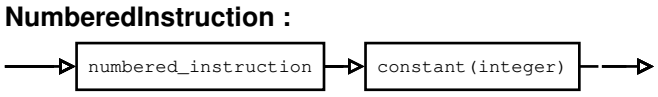

Figure 2.26: Michelson numbered instructions syntax.

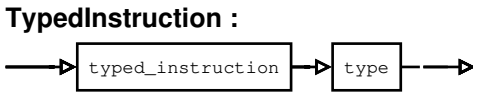

Figure 2.27: Michelson typed instructions syntax.

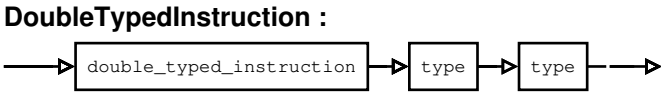

Figure 2.28: Michelson double typed instructions syntax.

As stated in this writing's introduction, Michelson code section consists on a sequence of semicolon separated instructions. This fact implies that Michelson lacks control statements, but, in exchange, Michelson does include control flow instructions. These instructions must include the code to run as parameters, either one or two parameters are needed, depending on the control statement the instruction is replacing, be it loop statements (Figure 2.29) or branch statements (Figure 2.30).

It is worth noting that these control flow instructions also include those needed to iterate over collections, such as the `MAP` or `ITER` instructions.

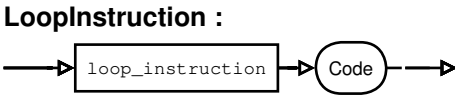

Figure 2.29: Michelson loop instructions syntax.

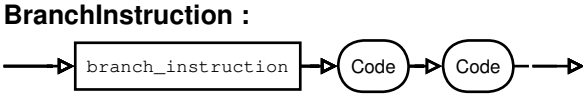

Figure 2.30: Michelson branch instructions syntax.

The last proper instruction class we will cover will be optionally numbered instructions (Figure 2.31). These instructions can be followed by a numeric parameter, which defaults to 1 if absent.

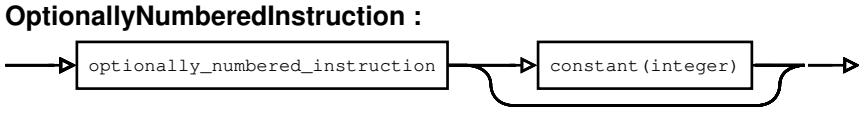

Figure 2.31: Michelson optionally numbered instructions syntax.

Special instructions and macros are two instruction classes which deserve a special mention, as their behaviour does not match those of the aforementioned instructions.

Firstly, special instructions (Figure 2.32) is a set comprising all the instructions that do not match any of the previously mentioned schemata.

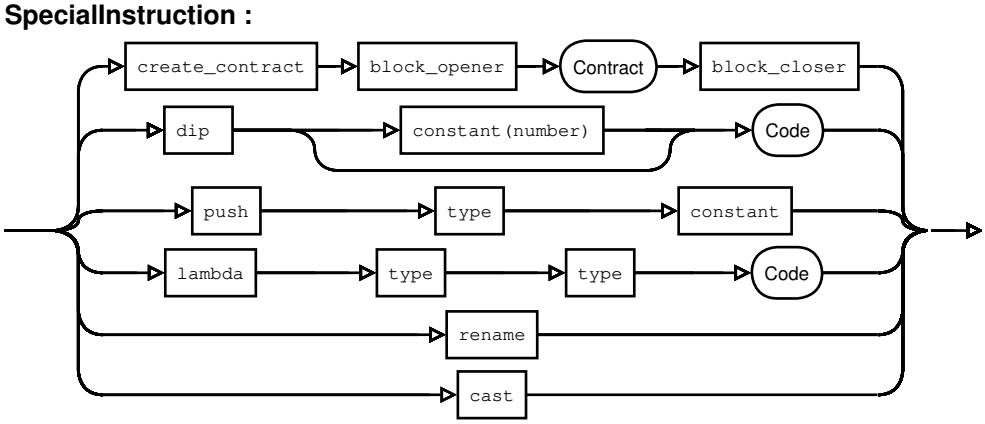

Figure 2.32: Michelson special instructions syntax.

And, last but not least, macros are those instructions which can be expressed as the concatenation of other simpler instructions. To better reproduce Michelson interpreter's behaviour, the parser deconstructs macros into its simplest representation, which may involve several iterations. This decision simplifies not only the analysis, but also type checking, as it makes the number of instructions to treat finite. Just like core instructions, macros are divided into similar classes, which behave exactly as instruction classes.

2.3. Type checking

Being Michelson a stack language, this type checking module has to check that each instruction's input stack belongs to the correct type. Which is to say, it has the proper number of elements and the types of these elements match those expected by the instruction.

The initial stack will contain only an element of type $(parameter, storage)$, and the output stack, a single element of type $(list(operation), storage)$. This process is complicated by the treatment of some instructions which need a special treatment.

The most obvious instruction in this list is the `LAMBDA` instruction, as the parser needs to check that the type output of the code matches the declared type for the given input

```
1   loop CODE / [bool|S] => S :- CODE / S => [bool|S].
```

Listing 2.1: LOOP instruction typing.

type. A similar case would be that of the CREATE_CONTRACT instruction, as this involves the creation of a new contract, passing its code section as a parameter. The main difference between these two instructions resides in the fact that lambdas implement functions that receive an input and return an output. This difference is portrayed in Equation 2.1 and Equation 2.2.

$$lambda\colon input \rightarrow output \tag{2.1}$$

$$contract\colon (parameter, storage) \rightarrow (list(operation), storage) \tag{2.2}$$

A similar case to this one would be that of branch and loop instructions. Simple loop instructions such as LOOP must not change the type of the stack, although they may alter its content. In order to test if a loop complies with this property, the type of the output stack on each iteration must be the same as that of the input stack, but with a new element of a certain type on top. In Listing 2.1, we can see a definition of one of these typing rules using a syntax similar to that of the implementation, simplified for the sake of clarity.

It is worth mentioning that, some loop instructions need to check a different condition, such as LOOP_LEFT, which checks whether the union type on top of the stack contains a left element. These instructions require its code to return a different type instead of bool on the top of the stack.

In the case of branch instructions, although the former restriction does not apply to them, a new one is introduced, as the type of the output stacks of both branches must match.

As it was pointed out in a previous section, there exist Michelson instructions to explicitly make a contract execution fail. These insert a special element on top of the stack, which represents the failure status. The type checking module must also treat this special case. In order to do so, the general rule states that every instruction may receive a failure stack as an input. If this is the case, the instruction does not alter the content of the stack.

In case of failure inside of a loop, the type checking module will accept this as a valid output type for the instruction. On the other hand, if failure occurs in one of the branches of a branch instruction, the output stack will be of type failure if and only if both branches fail. Otherwise, its type will be that of the non-failing branch.

As some Michelson instructions are polymorphic, the type checking module must also take into account this phenomenon. To achieve this goal, not only does this module check that the type of the input stack is correct, but it also outputs a specialized version of the instruction. This way, by making polymorphism transparent to the translation, analysis of the resulting program will be easier. In addition, it is worth mentioning that, thanks to this decision, the output of this module matches that of its counterpart found in Tezos code. This is both beneficial in terms of clarity, as it will make possible that the output instructions of the translation match the internal

instructions recognized by Tezos interpreter; and in terms of the precision of the analysis.

Finally, another service this module provides is constant type checking. This feature will be useful when parsing the code, as well as inputs used in Michelson interpreter.

2.4. Interpreter

Although not needed for the analysis, implementing an interpreter is useful to understand the semantics of a language and how resource consumption works.

The way this interpreter works is by dividing the code section into blocks of code. Each of these blocks is assigned a unique index, which will be used to refer to it by control flow instructions. This way, branches used in branch instructions, lambda functions present in the code or code blocks run by loop instructions are assigned an identifier.

Once a contract has been loaded, the interpreter runs its main block, passing the initial stack containing the parsed parameter and storage provided as arguments. Then, the interpreter proceeds to run the contract instruction by instruction.

During this process, each instruction is further divided into simpler operations, which in most cases simply consist on stack management tasks and a proper Prolog predicate containing the instruction semantics. The name of these predicates may differ from that of the internal operation output by the parser, due to the fact that the semantics and cost semantics of some of these operations are the same. For instance, the instruction `ADD` can be translated to one of several distinct operations, but, as some of them implement the same arithmetic expression and consume the same amount of *gas*, these are unified, including the predicate `add` in its definition.

As it can be seen in Equation 2.3, `ADD` can output different internal operations depending on the type of its parameters, but, as some of them behave exactly the same, their definitions will match.

$$ADD(A,B) \rightarrow \begin{cases} \left.\begin{array}{ll} add\_intint & \text{if } int(A), int(B) \\ add\_intnat & \text{if } int(A), nat(B) \\ add\_natint & \text{if } nat(A), int(B) \\ add\_natnat & \text{if } nat(A), nat(B) \end{array}\right\} \rightarrow add \\ \left.\begin{array}{ll} add\_timestamp\_to\_seconds & \text{if } timestamp(A), int(B) \\ add\_seconds\_to\_timestamp & \text{if } int(A), timestamp(B) \end{array}\right\} \rightarrow add\_timestamp \\ add\_tez \quad \text{if } mutez(A), mutez(B) \rightarrow add\_tez \end{cases} \tag{2.3}$$

In some cases, the only action carried out by an instruction is inserting or extracting elements from the stack. The definition of these includes a dummy instruction whose purpose will be treated in the following section.

For instance, the definition of the `LOOP code` instruction seen in Listing 2.2 includes a stack management operation to obtain the Boolean element on top of the stack and a

```
1 loop(Body) ==> pop(B), loop_(B,Body).
2
3 loop_(true,Body) ==> '$blt'('$loop'), '$run'(Body), pop(B),
     loop_(B,Body).
4
5 loop_(false,Body) ==> [].
```

Listing 2.2: LOOP instruction definition.

call to an auxiliary predicate, which will run the code block and recursively call itself in case the Boolean element equals true and do nothing otherwise.

2.5. Translator

In order to translate Michelson contracts to Horn Clauses, this module makes use of some of the features introduced in the previous section.

To begin the translation process, the translator parses the Michelson code in order to be able to load the different code blocks, assigning an id to each of them. Once the contract has been loaded, the translator obtains a Horn Clause representation of each of the blocks. This task poses a challenge to the translation process, as the translator has to abstract the Michelson stack in order to obtain a correct and easier to analyze Horn Clause representation.

In order to abstract Michelson's stack, the translator simply outputs those Horn Clauses that constitute the contract and its arguments, which proceed from a translation stack that is built through the translation process. This stack will contain every term appearing in the translation, be it constants introduced via PUSH instructions or other terms that would be present in Michelson stack, which are represented as variables.

The translator obtains each instruction definition by making use of the aforementioned mechanism implemented in the interpreter and then distinguishes stack management operations from those implementing the semantics, which will be represented as Horn Clauses in the final product. On the other hand, the former set of instructions will be used to manage the content of the translation stack, inserting and extracting the variables which will appear as input or output arguments in the predicates.

To better understand this process, it can be seen as if every stack element was assigned a term in the output program and stack management operations were the mechanism used to manage these variables.

As pairs are the most common and trivial to deal with data structure present in Michelson, this translation process also abstracts them. This decision makes analysis much easier, as it avoids the necessity to take into account the size of pair's elements.

As it was stated before, the translator has to output a Horn Clause representation for each block of source code. Until now, we have only explained how it deals with the main block, but, as we know, there are Michelson instructions implementing control

```
IF { FAIL } { NIL operation ; PAIR }
```

Listing 2.3: Michelson `IF` instruction example.

```
if__0(true,[A],B) :-
    '$const'('()'),
    failwith(['()',A],B).
if__0(false,[A],[(B,A)]) :-
    nil(B),
    '$cons_pair'.
```

Listing 2.4: `IF` instruction's translation.

flow operations which the translator has to deal with too. In order to do so, each control flow instruction is accompanied by extra information about the nature of its arguments, whether they are static or dynamic. Then, the translator only has to unify the static part and create a unique head for the resulting predicate by appending an auto increment value to the instruction name. Then, each predicate generated will receive two extra arguments: an input and an output stack.

As we can see, the Michelson instruction portrayed in Listing 2.3 implements a branch statement which will fail if the value on top of the stack is true and finish a contract execution otherwise. The translator's output for this instruction would be a call to the predicate defined in Listing 2.4. This predicate's head is formed by its name, the name of the instruction and unique identifier; the control flow condition and both the input and output stack.

A more complex example of this behaviour would be the treatment of loop instructions. In Listing 2.5, we can see a `MAP` instruction that runs a block of code for each element of a structure to produce a new structure containing the result of each execution. In this case, the instruction takes a list as an input, so the final instruction name would be `list_map`. The code block runs a lambda instruction in the stack using each element as an input and stores the result in the new list. As we can see in Listing 2.6, the head of the resulting predicate is formed by the name of the instruction name with a unique identifier, the original list, the resulting list and both the input and output stack. This predicate will call itself recursively and finish only when the list is empty.

2.6. Translation Example

In this section we will show the translation in action using an example. The contract to translate is the one displayed in Listing 2.7. As it can be seen, this contract contains control flow instructions such as `IF_CONS` or `ITER`, as well as macros. Due to this fact, the resulting translation will be too large to handle in one piece, so we have divided the original code in three logical sections.

The first section is used to access the parameter and set the stack up for the following section, which checks whether both vectors have the same length. Last but not least, the main section of the contract iterates over both lists to calculate the cross product which will be used by the last section to set the resulting contract storage.

```
MAP { DIP { DUP } ; EXEC }
```

Listing 2.5: Example of LOOP instruction.

```
list_map__0([],[],[A],[A]).
list_map__0([A|B],[C|D],[E],F) :-
    '$dip',
    '$dup',
    exec(E,A,C),
    list_map__0(B,D,[E],F).
```

Listing 2.6: LOOP instruction translation.

In order to better follow this process, the reader can refer to Listing A.1 and Listing B.1 in Appendices A and B respectively. The former contains type definitions of the used instructions, while the latter, the definition of the instructions itself. Throughout this section, unfolded version of the found macros are included to improve readability.

Before diving into the translation of the contract itself, the translation module generates a semi-static heading for the contract, shown in Listing 2.8. As it can be seen, this heading imports the needed packages and modules for the analysis and defines a regular type defining the possible contents of the returning stack. It is also worth noting the `entry` assertion that defines the types of the arguments used to call the contract and is used by CiaoPP's regular types domain.

Now, the first section of the code is quite straightforward, as it simply accesses the parameter and prepares the stack to check that both vectors have the same length. As it is shown in Listing 2.9, the UNPAIR macro is unfolded into the instructions that comprise it as seen in Equation 2.4. Due to the nature of the instructions found in this section, the result of its translation is comprised exclusively of dummy instructions.

$$\texttt{UNPAIR} \;/\; S \equiv \texttt{DUP} \;;\; \texttt{CAR} \;;\; \texttt{DIP}\;\{\texttt{CDR}\} \;/\; S \tag{2.4}$$

The second block of translated instructions (Listing 2.10) introduces the reader to how variables are managed during the translation process. For every new element introduced in the stack, the translation module declares a variable. In this example, it creates two variables, G and H, containing the length of the input lists. Then, these variables are used to check that both input vectors have the same dimensions with the instruction ASSERT_CMPEQ, which unfolds into several instructions as stated in Equation 2.5. In case they are not the same length, the call to the generated predicate `if__0` will cause the contract to fail, as this would violate the precondition of the contract.

$$\texttt{ASSERT\_CMPEQ} \;/\; S \equiv \texttt{COMPARE} \;;\; \texttt{EQ} \;;\; \texttt{IF}\;\{\}\;\{\;\texttt{UNIT} \;;\; \texttt{FAILWITH}\;\} \;/\; S \tag{2.5}$$

The reader may have noticed how the output instructions that calculate the length of the lists differ from those found in the original contract. This is due to the fact that SIZE is a polymorphic instruction and, as seen in Listing A.1 (Appendix A), the type

```
parameter (pair (list int) (list int)) ;
storage int ;
code { CAR ;
       DUP ;
       UNPAIR ;

       # Check that both vectors have the same length.
       SIZE ;
       DIP { SIZE } ;
       ASSERT_CMPEQ ;

       # Calculate cross product.
       UNPAIR ;
       DIP 2 { PUSH int 0 } ;
       ITER { SWAP ;
              IF_CONS { SWAP ; DIP { MUL ; ADD } }
                      # Never fails: vectors have same dimensions.
                      { FAILWITH }
            } ;
       DROP ;
       NIL operation ;
       PAIR }
```

Listing 2.7: A Michelson contract to calculate cross products.

```
:-module(A,[code/3,amount/1],[ciao_tezos(michelson_costs),regtypes]).
:-use_module(ciao_tezos(michelson_preds)).
:-regtype(return_stack/1).
return_stack([(A,B)]) :-
    list(operation,A),
    int(B).
return_stack(A) :-
    is_failed(A).
:-redefining(amount/1).
amount(A).
:-entry(code(A,B,C):(pair(A),int(B),var(C))).
```

Listing 2.8: Heading of the `Ciao` module containing the translated contract.

checking module selects different versions of this instruction depending on the type of the value on top of the stack.

Another important feature of this section to point out is the fact that variables `A` and `B`, containing both input vectors, are extracted from the stack when their length is calculated. But, as they had been previously duplicated in Listing 2.9, only one of their copies is lost, while the other one remains in the stack for the rest of the instructions to use. This is the reason why variables can appear more than once in a translated contract. Had they not been copied, they could have only been used by one instruction in the contract.

Another dummy instruction generated by the translation module for the analysis to

```
code((A,B),C,[(D,E)|F]) :-
    '$car',
    '$dup',
    '$dup',
    '$car',
    '$dip',
    '$cdr',
    ...
```

Listing 2.9: Section of the contract that accesses the parameters.

```
code((A,B),C,[(D,E)|F]) :-
    ...
    list_size(A,G),
    '$dip',
    list_size(B,H),
    compare_int(G,H,I),
    eq(I,J),
    '$if',
    if__0(J,[(A,B)],[(K,L)|M]),
    ...

if__0(true,[(A,B)],[(A,B)]).
if__0(false,[(A,B)],C) :-
    '$const'('()'),
    failwith(['()',(A,B)],C).
```

Listing 2.10: Section of the contract that checks whether both vectors have the same length.

calculate resource consumption is `'$if/0'`. This instruction is output by this module to account for the cost of executing the branch instruction. This is necessary, as each branch predicate generated by the translator will have its own name, so it would not be possible to include all of these in the cost model.

Lastly, the main section of the code, which calculates the cross product and returns the resulting stack is seen in Listing 2.11. This section performs three distinct actions.

To prepare the recursive operation that involves calculating the cross products, the first five instructions simply uncouple the input parameter and insert an integer value to accumulate the products of the elements of the vectors. As all of these instructions only manipulate the stack or access values in pairs, they are translated into dummy instructions for the analysis to take them into account, as their effects are abstracted during the translation process.

After this, the predicate implementing the instruction which iterates over the vector is called. This predicate, `list_iter_1/3`, receives two input arguments, the input list and the original stack; and returns the resulting stack as an output parameter. This predicate will call itself recursively until it has completely traversed the input list. Again, for the analysis to take into account the cost of each iteration, two dummy instructions are generated: `'$list_iter'` and `'$list_iter_end'`. These instructions

```
code((A,B),C,[(D,E)|F]) :-
    ...
    '$dup',
    '$car',
    '$dip',
    '$cdr',
    '$dipn'(2),
    '$const'(0),
    list_iter_1(K,[L,0|M],[N,E|F]),
    '$drop',
    nil(D),
    '$cons_pair'.

list_iter_1([],A,A) :-
    '$list_iter_end'.
list_iter_1([A|B],[C|D],E) :-
    '$list_iter',
    '$swap',
    '$if_cons',
    if_cons__2(C,[A|D],F),
    list_iter_1(B,F,E).

if_cons__2([],A,B) :-
    failwith(A,B).
if_cons__2([A|B],[C,D|E],[B,F|E]) :-
    '$swap',
    '$dip',
    mul(A,C,G),
    add(G,D,F).
```

Listing 2.11: Section of the code that implements a simple algorithm to calculate the cross product.

do not have any semantic meaning, but it is necessary that they differ from each other because the cost of starting a new iteration is greater than the cost of finishing the iteration process. As a result, these instructions will have a different associated cost in the cost model.

Lastly, the last section of the code simply modifies the stack for it to fit the expected format at contract exit.

Chapter 3

Michelson Contracts Analysis

In this chapter, we will cover the procedure to follow in order to obtain a precise and correct cost model to analyze Michelson contracts previously translated to Horn Clauses by using the translator covered in the previous chapter. In order to do so, we will firstly explain the different ways to obtain a cost model in Section 3.1, discussing the pros and cons of each of them. Then, in Section 3.2, we will dive into our cost model and explain the resources included in it.

3.1. Obtaining a cost model

A cost model is a definition of the resource semantics of a language. It is necessary to have a cost model in order to perform cost analysis on a program, as this will contain the rules the analyzer will follow. As a result, the better a model reflects the resource semantics of the platform used, the more precise the analysis will be.

In the present case, we are studying virtual resources, so there is no need to measure the performance of real machines running the code. Instead, there are other ways to obtain a cost model, each of which have their pros and cons we will discuss in this section.

First of all, the most obvious way to obtain a precise cost model is using an already existent definition, provided by the platform under study. This would be the cheapest and most precise way to do so, as this model would contain the same set of rules used by the platform itself and would not need to study the code or make measurements in order to obtain these rules. On the other hand, not all platforms count with a definition of their resource semantics, as it is the case with Tezos, so another solution has to be considered in those cases.

As we are dealing with virtual resources, we know there must exist sections of the code implementing those resource semantics. In this case, another way to obtain a model would be to perform program slicing on the code to obtain these rules. Slicing is a technique used to extract relevant statements of the code following a criterion, in our case, we would have to extract the statements which specify how the resources studied are consumed. By using this technique, we would obtain a highly precise cost model, as we would be studying the very same code that dictates how resources are consumed, but, at the same time, it would be more expensive than the former

method, as it would be necessary to develop the tool used to perform the program slicing.

Following the same principles as in the former method, another way to obtain the cost model would be to analyze the same code using an analyzer such as `CiaoPP`. The result of this method would be similar to the former, but it would be necessary to count with a powerful enough analyzer supporting the language used in the code. In our case, Tezos source code is written in Ocaml, a language not supported by `CiaoPP`, so this method is not feasible.

The last method we will discuss is manual extraction of the cost model from the source code. This method can be cumbersome to say the least, as it may involve inspecting hundreds of lines of code and it may lead to confusion and mistakes. As previously stated, Tezos source code is written in Ocaml, so, in our case, it would also involve learning a new language. On the other hand, this method can be faster when not counting with the aforementioned conveniences, due to the fact that it is possible to dive in the code and find the relevant statements directly.

This last method is the one used in this work. Being the Tezos platform open source, we simply had to access their GitLab repository, learn how Ocaml and the libraries they used worked and find the relevant statements to obtain the cost model. We learned how Tezos uses a different cost model on each of their protocol versions, which increases the value of counting with a configurable tool such as `CiaoPP`. Although it was a complicated process, we also got some information which was not present or clear enough in Tezos literature, so going through Tezos source code was valuable in more ways than obtaining a cost model.

3.2. Tezos Cost Model

As previously explained, this Tezos cost model was obtained by inspecting the platform's source code. Throughout this process, we were able to learn more about how the platform works and, more specifically, how *gas* consumption in Tezos platform works.

In this project, we are analyzing contracts in Tezos' latest protocol version at the time of writing, *Carthage*; so the code visited to obtain the model was the one belonging to this latest implementation. Despite this fact, it is worth noting that obtaining a cost model for any of the previous versions of the platform, or even for other platforms using Michelson as a smart contracts language, such as Dune, would be similar.

In this section, we will explain how we identified each resource and its nature in our cost model in Subsection 3.2.1 and how we included each instruction to study in the cost model in Subsection 3.2.2.

3.2.1. Resources

The first thing noted when inspecting the code was that the resource to study, *gas*, was not an atomic resource. In other words, it is a compound resource which can

```
:- resource michelson_allocations.
:- resource michelson_steps
:- resource michelson_reads.
:- resource michelson_writes.
:- resource michelson_bytes_read.
:- resource michelson_bytes_written.
```

Listing 3.1: Assertions to declare the resources to study.

```
:- default_cost(ub,michelson_steps,0).
:- default_cost(lb,michelson_steps,0).
:- head_cost(lb,michelson_steps,0).
:- head_cost(ub,michelson_steps,0).
:- literal_cost(lb,michelson_steps,0).
:- literal_cost(ub,michelson_steps,0).
:- trust_default + cost(lb,michelson_steps,0).
:- trust_default + cost(ub,michelson_steps,0).
```

Listing 3.2: Assertions to declare the default cost of a resource.

be expressed in terms of other resources, as expressed in Equation 3.1. As a consequence, each Michelson instruction will consume one or several of these resources, which will later be used to calculate *gas* consumption.

$$gas(allocations, steps, reads, writes, bytes\_read, bytes\_written) = \\ 2^{-7} * \begin{pmatrix} allocations \\ steps \\ reads \\ writes \\ bytes\_read \\ bytes\_written \end{pmatrix} \times \begin{pmatrix} 2 \\ 1 \\ 100 \\ 160 \\ 10 \\ 15 \end{pmatrix} \tag{3.1}$$

In order to include these resources in our cost model, we have to use `Ciao` assertions, as those included in Listing 3.1. As we can see, these only have to include the name of the resource to be studied for it to be included in the analysis.

Now, as in most cases not all resources will be consumed by every instruction, our model has to include default cost assertions to avoid having to express that fact with cost assertions. Listing 3.2, shows the assertions used in the model to set the default cost for resource *michelson_steps* to 0. This very same assertions can be used to apply the same default cost to the other resources.

As we stated before, *gas* is a compound resource which can be expressed in terms of other resources, as a result, *gas* has to be included in our model too. In Listing 3.3, we can see how *gas* is declared not only as a resource in our model, but also as a compound resource expressed in terms of the other resources found in the model following the expression in Equation 3.1.

```
:- resource gas.

:- compound_resource(gas, exp(2,-7) * (
    2 * michelson_allocations +
    michelson_steps +
    100 * michelson_reads +
    160 * michelson_writes +
    10 * michelson_bytes_read +
    15 * michelson_bytes_written
)).
```

Listing 3.3: Assertions to declare *gas* as a resource.

3.2.2. Instructions

Once the resources to be inferred by the analysis have been included in the cost model, we can proceed to also declare the cost of Michelson instructions in terms of those resources.

To illustrate the steps followed throughout this process, we will explain how the cost of the `PUSH type value` instruction was obtained. Firstly, it is worth mentioning that the internal representation of this instruction is `Const v`, as neither the instruction semantics nor its cost depends on the type of the constant pushed into the stack. In Listing 3.4, we can see the definition of this basic operation in the code of the Michelson interpreter. As it can be seen, this code snippet contains not only the semantics of the instruction, but also its cost semantics, although not explicitly stated.

```
(Const v, rest) ->
      Lwt.return (Gas.consume ctxt Interp_costs.push)
      >>=? fun ctxt -> logged_return (Item (v, rest), ctxt)
```

Listing 3.4: `PUSH type value` resource semantics.

In order to decipher the meaning of the cost expression found in `Const v` definition, we have to refer to the code fragment present in Listing 3.5. This very same cost expression will be used by several other instructions, so it is not necessary to take this step more than once. In this case, as it can be seen, the cost of a `push` operation is expressed in terms of a function, `atomic_step_cost`, displayed in Listing 3.6. This function, as well as other similar functions expressing operation costs, returns an object accounting the amount of each resource this instruction consumes. In this case, each `push` operation consumes 20 allocations, which is a base resource in our cost model; so each `PUSH type value` instruction accounts for 20 allocations in our cost model.

```
let push = atomic_step_cost 10
```

Listing 3.5: `push` Michelson cost.

```
let atomic_step_cost n =
{
  allocations = Z.zero;
  steps = Z.of_int (2 * n);
  reads = Z.zero;
```

```
:- trust pred '$const'(A)
   => gnd(A)
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_steps,20),cost(ub,michelson_steps,20)
     ).
```

Listing 3.7: push Michelson cost.

```
  writes = Z.zero;
  bytes_read = Z.zero;
  bytes_written = Z.zero;
}
```

Listing 3.6: atomic_step_cost definition.

Once the cost of an instruction has been obtained, it is time to include it in our model. As we are using CiaoPP, this operation involves using Ciao assertions. To be more precise, as PUSH type value translates to a dummy instruction, we have to express the cost of this dummy instruction. This assertion, shown in Listing 3.7, includes information about the typing of the arguments of the instruction, => gnd(A) states that the only input argument will be ground after running the instruction. Every assertion used in this model also provides information about the cardinality of Michelson instructions, as this is crucial to perform a correct resource analysis. In this case, every Michelson instruction is a deterministic function defined in all of its domain which does not fail. This fact greatly simplifies Michelson analysis. Last but not least, the cost of the instruction is expressed, giving both an upper and lower bound for CiaoPP to calculate both approximations. As for this instruction, this cost is constant, but, as we will see, this cost may be expressed in terms of the input parameters.

Now, we can apply this same steps to obtain the cost of a more complex instruction: ADD. As this is a polymorphic instruction, so it may output different predicates in the translation process. In this case, we will focus on the instance of this instruction capable of dealing with integers and natural numbers, which was called add in Equation 2.3.

```
  | (Add_intint, Item (x, Item (y, rest))) ->
      consume_gas_binop descr (Script_int.add, x, y)
        Interp_costs.add rest ctxt
  | (Add_intnat, Item (x, Item (y, rest))) ->
      consume_gas_binop descr (Script_int.add, x, y)
        Interp_costs.add rest ctxt
  | (Add_natint, Item (x, Item (y, rest))) ->
      consume_gas_binop descr (Script_int.add, x, y)
        Interp_costs.add rest ctxt
  | (Add_natnat, Item (x, Item (y, rest))) ->
      consume_gas_binop
        descr
        (Script_int.add_n, x, y)
        Interp_costs.add
        rest
        ctxt
```

Listing 3.8: Some of ADD's definitions.

As it can be seen comparing Equation 2.3 and Listing 3.8, our translation process complies with Tezos internal representation of Michelson instructions. All of these internal operations implement the same function and consume the same amount of *gas*, expressed in terms of the add function, which takes both arguments of the addition operation as input parameters.

```
let add i1 i2 =
  atomic_step_cost
    (51 + (Compare.Int.max (int_bytes i1) (int_bytes i2) / 62))
```

Listing 3.9: add's cost definition.

If we refer to this function, shown in Listing 3.9, we can see how the same atomic_step_cost function is used to express the cost of the operation, but this time the cost is expressed in terms of the arguments of the instruction. The definition of int_bytes is not included in Tezos source code, but it can be inferred from some comments present in the repository and is expressed as a mathematical function in Equation 3.2.

$$int\_bytes(x) = 1 + \left\lfloor \frac{\log_2 |x|}{8} \right\rfloor \tag{3.2}$$

Using this and previous knowledge, we can simplify the cost expression to that of Equation 3.3. This way, we can optimize our cost model by using simple and correct arithmetic.

$$\begin{aligned} cost_{add}(A,B) &= 2 * \left(51 + \frac{max\left(1 + \left\lfloor \frac{\log_2 |A|}{8} \right\rfloor, 1 + \left\lfloor \frac{\log_2 |B|}{8} \right\rfloor\right)}{62} \right) \\ &= 102 + \frac{1 + \left\lfloor \frac{\log_2 max(|A|,|B|)}{8} \right\rfloor}{31} \end{aligned} \tag{3.3}$$

As in the previous example, in order to include this instruction in our cost model, we have to write a cost assertion. In Listing 3.10, we can see the aspect of this assertion, which expresses the exact cost of this instruction in terms of its inputs. As we can see, again, we state the type of the arguments of the predicate at its exit and other relevant information about its cardinality. The notation used in the arithmetic expression that defines the cost of the instruction closely resembles that used in Equation 3.3, which contributes to the readability of this model.

```
:- trust pred add(A,B,C)
  => ( int(A), int(B), int(C) )
  + ( not_fails, covered, i_det, cardinality(1,1),
      cost(lb,michelson_steps,102 + (1 +
         log2(max(int(A),int(B)))/8)),
      cost(ub,michelson_steps,102 + (1 +
         log2(max(int(A),int(B)))/8))
    ).
```

Listing 3.10: add's cost assertion.

Chapter 4

Analysis Examples

This chapter includes some analysis examples using the cost model obtained in the previous chapter. Firstly, Section 4.1 includes the results of analyzing a contract whose cost depends on the value of the input integer parameters. Then, Section 4.2 analyzes a contract whose cost is a linear function with respect to the length of the input list. Finally, in Section 4.3, these results are presented in a table for the sake of clarity.

4.1. Michelson contract analysis: `michelson_arithmetics`

In this section we will analyze the contract shown in Listing 4.1, which implements the following function:

$$f(Parameter, Storage) = Parameter^2 + 2 * Storage + 1 \tag{4.1}$$

This makes its analysis of great interest, as the cost of arithmetic operations on integers of arbitrary length depends on the value of their operands. This phenomenon is better explained in Equation 4.2, which expresses the cost of running the contract in terms of the input arguments using the expressions in Equation 4.3 and Equation 4.4. The constant value added to the cost of running the arithmetic operations corresponds to that of the previous instructions, which present a constant cost not dependent on the input values.

$$cost(A, B) = k + cost_{mul}(A, A) + cost_{mul}(2, \tag{4.2}$$

$$cost_{add}(A, B) = 102 + log_2(max(A, B))/248 \tag{4.3}$$

$$cost_{mul}(A, B)! = 102 + (log_2(max(A, B)) + 8) * log_2(log_2(max(A, B))/8 + 1) \tag{4.4}$$

As seen in Listing C.1 (Appendix C), the cost model not only provides `CiaoPP` with information about the cost of each predicate in terms of the declared resources, but also with information about the size of the output parameters of each predicate. This way, the analysis tool will be able to infer the cost of the instructions in lines 7 and

```
1  parameter int ;
2  storage int ;
3  code { UNPAIR ;
4         DUP ;
5         MUL ;
6         DIP { PUSH int 2 ; MUL } ;
7         ADD ;
8         PUSH int 1 ;
9         ADD ;
10        NIL operation ;
11        PAIR }
```

Listing 4.1: Michelson contract with $O(1)$ complexity.

```
1  :-module(A,[code/3,amount/1],[ciao_tezos(michelson_costs),regtypes]).
2  :-use_module(ciao_tezos(michelson_preds)).
3  :-regtype(return_stack/1).
4  return_stack([(A,B)]) :-
5      list(operation,A),
6      int(B).
7  return_stack(A) :-
8      is_failed(A).
9  :-redefining(amount/1).
10 amount(A).
11 :-entry(code(A,B,C):(int(A),int(B),var(C))).
12 code(A,B,[(C,D)]) :-
13     '$dup',
14     '$car',
15     '$dip',
16     '$cdr',
17     '$dup',
18     mul(A,A,E),
19     '$dip',
20     '$const'(2),
21     mul(2,B,F),
22     add(E,F,G),
23     '$const'(1),
24     add(1,G,D),
25     nil(C),
26     '$cons_pair'.
```

Listing 4.2: Horn Clauses representation of the contract in Listing 4.1.

9, whose input arguments are the result of running the instructions in lines 5 and 6, without analyzing the implementation of the Michelson predicates.

The result of the translation of the present contract can be seen in Listing 4.2. Here, we can see more clearly how the input parameters of the predicates in lines 22 and 24 come from the output of those in lines 18 and 21.

Listing 4.3 displays the result of running CiaoPP on the present contract. This output

```
:- true pred code(A,B,C)
   : ( int(A), int(B), var(C) )
   => ( int(A), int(B), rt2(C), size(ub,A,int(A)),
      size(ub,B,int(B)), size(ub,C,1) )
   + ( cost(ub,michelson_allocations,0),
       cost(ub,michelson_bytes_read,0),
       cost(ub,michelson_bytes_written,0),
       cost(ub,michelson_gas,
           % Cost of running mul(2,B,F)
           0.00032552083333333333*(log(2,log(256,int(B)+2)+1)*
               log(2,256*int(B)+512))+
           % Cost of running mul(A,A,E)
           0.00032552083333333333*(log(2,log(256,int(A))+1)*
               log(2,256*int(A)))+
           % Cost of running add(1,G,D)
           0.0009765625*log(2,exp(int(A),2)+2*int(B)+1)+
           % Cost of running add(E,F,G)
           0.0009765625*log(2,exp(int(A),2)+2*int(B))+
           5.078125),
       cost(ub,michelson_reads,0),
       cost(ub,michelson_steps,
           % Cost of running mul(2,B,F)
           0.041666666666666664*(log(2,log(256,int(B)+2)+1)*
               log(2,256*int(B)+512))+
           % Cost of running mul(A,A,E)
           0.041666666666666664*(log(2,log(256,int(A))+1)*
               log(2,256*int(A)))+
           % Cost of running add(1,G,D)
           0.125*log(2,exp(int(A),2)+2*int(B)+1)+
           % Cost of running add(E,F,G)
           0.125*log(2,exp(int(A),2)+2*int(B))+
           650),
       cost(ub,michelson_writes,0) ).
```

Listing 4.3: Result of running the analyzer on the contract in Listing 4.2.

assertion, indented and commented for the sake of clarity, shows how the abstract domain used by the analysis tool infers the cost of each arithmetic instruction correctly, even that of the instructions whose cost depends on the output of previous operations. This expression clearly resembles Equation 4.2, which proves the correctness of the result of the analysis.

4.2. Michelson contract analysis: `list_map`

In this section, we will show the result of analyzing a previously translated contract using a user defined Tezos cost model on `CiaoPP`.

The contract to analyze is shown in Listing 4.4. This contract receives a mutez value, which is to say, a 64-bit integer, and adds that number to each value in its storage,

which is a list of mutez itself. As a consequence, it presents $O(n)$ complexity.

```
parameter mutez ;
storage (list mutez) ;
code { UNPAIR ;
       SWAP ;
       MAP { DUP 2 ; ADD } ;
       DIP { DROP } ;
       NIL operation ;
       PAIR }
```

Listing 4.4: Contract with $O(n)$ complexity.

As in the previous example, the first step to analyze this contract would be to obtain its Horn Clauses translation by using the translation tool described in the previous section. This translation is shown in Listing 4.5, where it can be seen that there is an almost one-to-one correspondence between Michelson instructions and predicates, except for expanded macros such as `UNPAIR` and other variations better explained in Chapter 2.

This `Ciao` module exhibits the exact same behaviour as the contract in Listing 4.4, as it consists mainly on a call to a predicate which explores the storage list recursively, adding the input parameter to each element in the collection and returning the altered structure. Each iteration on the list will have a constant cost, as every predicate in the loop has a constant code. And every predicate in the main block of the contract has a constant cost, so the cost expression of this contract will be something similar to that in Equation 4.5.

$$cost(Parameter, Storage) = a * length(Storage) + b \tag{4.5}$$

If `CiaoPP` is run on the translated contract, it will output an upper limit for the cost of running the contract, which will be exactly the same as the actual cost. As seen in Listing 4.6, the cost in terms of *gas* of running this contract is linear to the length of the storage, which is the second argument used to call the contract; so the shape of the output of the analysis is similar to that of Equation 4.5.

In addition to that, it is also possible to obtain the cost of running the recursive predicate. In Listing 4.7 we can see the output of the analysis, which also follows a linear expression. This is due to the fact that the analyzer does not output the cost of running a single iteration of the predicate, but the cost of all the iterations. It can be seen that this predicate also has a base constant cost in case of running it with an empty list as an input parameter.

By subtracting the expression obtained in Listing 4.7 to that in Listing 4.6, we could obtain the cost of running those instructions in the main block of the contract. If we refer to Listing C.1 in Appendix C, where a stripped version of the cost model showing only those instructions displayed in the example is found, we can check the correctness of the output of the analysis.

```
:-module(A,[code/3,amount/1],[ciao_tezos(michelson_costs),regtypes]).
:-use_module(ciao_tezos(michelson_preds)).
:-regtype(return_stack/1).
return_stack([(A,B)]) :-
    list(operation,A),
    list(mutez,B).
return_stack(A) :-
    is_failed(A).
:-redefining(amount/1).
amount(A).
:-entry(code(A,B,C):(mutez(A),list(mutez,B),var(C))).
code(A,B,[(C,D)|E]) :-
    '$dup',
    '$car',
    '$dip',
    '$cdr',
    '$swap',
    list_map__0(B,D,[A],[F|E]),
    '$dip',
    '$drop',
    nil(C),
    '$cons_pair'.
list_map__0([],[],[A],[A]) :-
    '$list_map_end'.
list_map__0([A|B],[C|D],[E],F) :-
    '$list_map',
    '$dip',
    '$dup',
    '$swap',
    add_tez(E,A,C),
    list_map__0(B,D,[E],F).
```

Listing 4.5: Horn Clauses translation of the contract.

```
:- true pred code(A,B,C)
   : ( mutez(A), list(mutez,B), var(C) )
   => ( mutez(A), list(int,B), rt28(C),
        size(ub,A,int(A)), size(ub,B,length(B)), size(ub,C,inf) )
   + ( cost(ub,michelson_allocations,0),
       cost(ub,michelson_bytes_read,0),
       cost(ub,michelson_bytes_written,0),
       cost(ub,michelson_gas,2.203125*length(B)+2.1875),
       cost(ub,michelson_reads,0),
       cost(ub,michelson_steps,282*length(B)+280),
       cost(ub,michelson_writes,0)
     ).
```

Listing 4.6: Analysis output for the main predicate.

```
:- true pred list_map__0(_A,_B,_C,F)
   : ( list(mutez,_A), var(_B), rt4(_C), list_functor(F) )
   => ( list(int,_A), list(int,_B), rt4(_C), rt4(F),
        size(ub,_A,length(_A)), size(ub,_B,length(_A)),
        size(ub,_C,1), size(ub,F,1) )
   + ( cost(ub,michelson_allocations,0),
       cost(ub,michelson_bytes_read,0),
       cost(ub,michelson_bytes_written,0),
       cost(ub,michelson_gas,2.203125*length(_A)+0.46875),
       cost(ub,michelson_reads,0),
       cost(ub,michelson_steps,282*length(_A)+60),
       cost(ub,michelson_writes,0)
     ).
```

Listing 4.7: Analysis output for the recursive predicate.

4.3. Table of results

In this section, we summarize the previously obtained results, adding valuable information about the contracts and the analysis process.

Table 4.1 shows the previously discussed results, adding information about the size in bytes of the original Michelson code and the size of its Horn Clauses representation obtained after running the translation tool. As it can be seen, the size of the translated contracts is larger than the original contract. This is due to the fact that the translation process involves altering the original contract, introducing variables and declaring a predicate for each block of code in the original Michelson contract. This drawback is justified, as the resulting product is easier to analyze, which is the main purpose of this work.

Contract	C. Size (B)	Translated C. Size (B)	Resource A. gas	A. Time (ms)
michelson_arithmetics	189	538	$log(\alpha^2 + 2 * \beta)$	544.96
list_map	159	691	β	660.5
addition	258	533	$log(\alpha)$	644.847
bytes	240	752	$log(\beta)$	831.632
apply	218	546	k	507.38

Table 4.1: Table of results of the analysis.

Table 4.1 also collects the complexity orders of the obtained expressions when running the analysis in terms of its input parameters. In this case, α denotes the size of the input parameter and β, the size of the storage. These sizes will depend on the metrics associated to the input parameters, which is to say, if the input parameter is a list, the size metric used will be the length of the list. On the other hand, if the input parameter is an integer number, the size metric used will be its absolute value. This table also shows the required time to run the analysis, which includes loading the module to analyze to parse the present assertions and analyzing the contract in different abstract domains to obtain the desired result.

Contract	Resource A.			
	steps	allocations	writes	written_bytes
`michelson_arithmetics`	$log(\alpha^2 + 2 * \beta)$	-	-	-
`list_map`	β	-	-	-
`addition`	$log(\alpha)$	-	k	k
`bytes`	$log(\beta)$	-	k	k
`apply`	k	k	k	k

Table 4.2: Inferred cost in terms of Michelson atomic resources.

The first two contracts displayed can be found in the previous sections of this chapter, whereas the rest can be found in Appendices D, E and F respectively. In these appendices, the reader will find the original Michelson contract, its Horn Clauses representation and a snippet taken from the output of the analysis tool.

Chapter 5

Conclusion

To put everything in a nutshell, it is possible to tackle smart contracts analysis using a generic approach, just by implementing a simple tool to obtain the desired representation of the programs. In this work, we have managed to implement a Michelson to Horn Clause translator and to express a cost model in the `Ciao` assertion language, which allow `CiaoPP` to perform a static analysis on Tezos contracts. This approach can be of great help in a rapidly changing environment in which new languages arise within months and cost models suffer alterations with each platform iteration. As the results of our experimental assessment conclude, not only is this analysis working on the unaltered output of the automatic translation tool for the selected benchmarks, but also it is performing arithmetic operations on the cost expressions of different resources to obtain the wanted *gas* cost function. These results, whose correctness and accuracy can be easily verified by inspecting our Michelson cost model, were obtained within a reasonable time given the size of the benchmarks, which compels one to think that this is a very promising approach.

Despite the successful outcome of the assessment, there are some cases in which the analysis did not work as expected, such as non-trivial control flow instructions or contracts whose input contains nested data structures. In order to overcome this challenge, it will be necessary to keep working on the translation tool, exploiting Michelson limitations to apply techniques such as stack deforestation when dealing with control flow instructions or further deconstructing data structures in order to facilitate the analysis. Even so, the accuracy of this analysis together with the fact that it is completely parametric and configurable by the user notes that we are hopefully on the right track.

In conclusion, our proposed approach to the static analysis of smart contracts is feasible, can be accurate and efficient, and is indeed a promising avenue for future research.

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." 2008.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger." Gavin Wood's original Ethereum paper, 2016.

[3] V. Allombert, M. Bourgoin, and J. Tesson, "Introduction to the tezos blockchain," *CoRR*, vol. abs/1909.08458, 2019.

[4] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[5] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "GASTAP: A gas analyzer for smart contracts," *CoRR*, vol. abs/1811.10403, 2018.

[6] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: gas analysis and optimization for ethereum smart contracts," *CoRR*, vol. abs/1912.11929, 2019.

[7] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: surviving out-of-gas conditions in ethereum smart contracts," *PACMPL*, vol. 2, no. OOPSLA, pp. 116:1–116:27, 2018.

[8] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia, "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)," *Science of Computer Programming*, vol. 58, pp. 115–140, October 2005.

[9] M. Méndez-Lojo, J. Navas, and M. Hermenegildo, "A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs," in *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, no. 4915 in Lecture Notes in Computer Science, pp. 154–168, Springer-Verlag, August 2007.

[10] J. Navas, E. Mera, P. Lopez-Garcia, and M. Hermenegildo, "User-Definable Resource Bounds Analysis for Logic Programs," in *23rd International Conference on Logic Programming (ICLP'07)*, vol. 4670 of *Lecture Notes in Computer Science*, Springer, 2007. 10-year Test of Time Award.

[11] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, and G. Puebla, "An Overview of Ciao and its Design Philosophy," *Theory and Practice of Logic Programming*, vol. 12, pp. 219–252, January 2012. http://arxiv.org/abs/1102.5497.

[12] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo, "Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption," *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification*, vol. 18, pp. 167–223, March 2018. arXiv:1803.04451.

[13] S. K. Debray, N.-W. Lin, and M. V. Hermenegildo, "Task Granularity Analysis in Logic Programs," in *Proc. 1990 ACM Conf. on Programming Language Design and Implementation (PLDI)*, pp. 174–188, ACM Press, June 1990.

[14] S. K. Debray and N. W. Lin, "Cost Analysis of Logic Programs," *ACM Transactions on Programming Languages and Systems*, vol. 15, pp. 826–875, November 1993.

[15] S. K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin, "Lower Bound Cost Estimation for Logic Programs," in *1997 International Logic Programming Symposium*, pp. 291–305, MIT Press, Cambridge, MA, October 1997.

[16] A. Serrano, P. Lopez-Garcia, and M. V. Hermenegildo, "Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types," *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, vol. 14, pp. 739–754, July 2014.

[17] J. Navas, M. Méndez-Lojo, and M. Hermenegildo, "Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications," in *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pp. 29–32, April 2008. Extended Abstract.

[18] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder, "Energy Consumption Analysis of Programs based on XMOS ISA-level Models," in *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers* (G. Gupta and R. Peña, eds.), vol. 8901 of *Lecture Notes in Computer Science*, pp. 72–90, Springer, 2014.

[19] Y. Lichtenstein and E. Y. Shapiro, "Abstract algorithmic debugging," in *Fifth International Conference and Symposium on Logic Programming* (R. A. Kowalski and K. A. Bowen, eds.), (Seattle, Washington), pp. 512–531, MIT, August 1988.

[20] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Programming Languages Design and Implementation'93*, pp. 46–55, 1993.

[21] M. Comini, G. Levi, and G. Vitiello, "Declarative diagnosis revisited," in *1995 International Logic Programming Symposium*, (Portland, Oregon), pp. 275–287, MIT Press, Cambridge, MA, December 1995.

[22] M. Comini, G. Levi, M. C. Meo, and G. Vitiello, "Abstract diagnosis," *Journal of Logic Programming*, vol. 39, no. 1–3, pp. 43–93, 1999.

[23] P. Cousot, "Automatic Verification by Abstract Interpretation, Invited Tutorial," in *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, no. 2575 in LNCS, pp. 20–24, Springer, January 2003.

[24] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski, and G. Puebla, "On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs," in *Proc. of the 3rd*

Int'l. Workshop on Automated Debugging–AADEBUG'97, (Linköping, Sweden), pp. 155–170, U. of Linköping Press, May 1997.

[25] M. V. Hermenegildo, G. Puebla, and F. Bueno, "Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging," in *The Logic Programming Paradigm: a 25–Year Perspective* (K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, eds.), pp. 161–192, Springer-Verlag, July 1999.

[26] C. Flanagan, "Hybrid Type Checking," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006* (J. G. Morrisett and S. L. Peyton Jones, eds.), pp. 245–256, ACM, 2006.

[27] M. Fähndrich and F. Logozzo, "Static Contract Checking with Abstract Interpretation," in *Int'l. Conf. on Formal Verification of Object-oriented Software, FoVeOOS'10*, vol. 6528 of *LNCS*, pp. 10–30, Springer, 2011.

[28] S. Tobin-Hochstadt and D. Van Horn, "Higher-Order Symbolic Execution via Contracts," in *OOPSLA*, pp. 537–554, ACM, 2012.

[29] P. Nguyen and D. V. Horn, "Relatively Complete Counterexamples for Higher-Order Programs," in *PLDI'15*, pp. 446–456, ACM, 2015.

[30] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo, "User-Definable Resource Usage Bounds Analysis for Java Bytecode," in *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, vol. 253 of *Electronic Notes in Theoretical Computer Science*, pp. 65–82, Elsevier - North Holland, March 2009.

[31] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko, "HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution)," in *TACAS* (C. Flanagan and B. König, eds.), vol. 7214 of *LNCS*, pp. 549–551, Springer, 2012.

[32] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, "A Verification Toolkit for Numerical Transition Systems - Tool Paper," in *Proc. of FM 2012*, vol. 7436 of *LNCS*, pp. 247–251, Springer, 2012.

[33] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[34] N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, eds., *Workshop on Horn Clauses for Verification and Synthesis*, July 2014. Electronic Proceedings in Theoretical Computer Science.

[35] B. Kafle, J. P. Gallagher, and J. F. Morales, "RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I* (S. Chaudhuri and A. Farzan, eds.), vol. 9779 of *Lecture Notes in Computer Science*, pp. 261–268, Springer, 2016.

Appendix A

Typing rules of the instructions

```
failwith::failwith / [A|_] => [failed(A)].
if(Bt,Bf)::if(IBt,IBf) / [bool|A] => B.
dip(Code)::dip(ICode) / [Top|A] => [Top|B].
dipn(N,Code)::dipn(N,ICode) / A => B.
drop::drop / [_Top|S] => S.
dup::dup / [Top|S] => [Top,Top|S].
swap::swap / [A,B|C] => [B,A|C].
push(Ta,V)::const(V) / S => [Ta|S].
unit::const('()') / S => [unit|S].
eq::eq / [int|S] => [bool|S].
compare::compare(Ty) / [Ty,Ty|S] => [int|S].
add::add_intint / [int,int|S] => [int|S].
mul::mul_intint / [int,int|S] => [int|S].
pair::cons_pair / [Ta,Tb|S] => [pair(Ta,Tb)|S].
car::car / [pair(Ta,_)|S] => [Ta|S].
cdr::cdr / [pair(_,Tb)|S] => [Tb|S].
size::list_size / [list(_)|S] => [nat|S].
iter(Body)::list_iter(IBody) / [list(Ty)|S] => S.
nil(T)::nil / S => [list(T)|S].
if_cons(Bt,Bf)::if_cons(IBt,IBf) / [list(Ta)|A] => B.
```

Listing A.1: Simplified typing rules of the instructions used in the example.

Appendix B

Definition of the instructions

```
failwith ==> '$blt_st'(failwith).

if(Bt,Bf) ==> '$blt_st'(pop(B)), '$blt'('$if'), if_(B,Bt,Bf).
if_(true,Bt,_Bf) ==> '$run'(Bt).
if_(false,_Bt,Bf) ==> '$run'(Bf).

dip(Code) ==> '$blt'('$dip'), '$blt_st'(pop(Top)), '$run'(Code),
    '$blt_st'(push(Top)).

dipn(N,Code) ==> '$blt'('$dipn'(N)), '$dipn'(N,['$run'(Code)]).
'$dipn'(0,Insns) ==> '$run0'(Insns) :- !.
'$dipn'(N,Insns) ==>
    '$blt_st'(pop(Value)), '$dipn'(M,Insns), '$blt_st'(push(Value))
       :-
        M is N - 1.

drop ==> '$blt'('$drop'), '$blt_st'(pop(_)).

dup ==>
    '$blt'('$dup'), '$blt_st'(pop(X)), '$blt_st'(push(X)),
       '$blt_st'(push(X)).

swap ==>
    '$blt'('$swap'), '$blt_st'(pop(X)), '$blt_st'(pop(Y)),
    '$blt_st'(push(X)), '$blt_st'(push(Y)).

const(Value) ==> '$blt'('$const'(Value)), '$blt_st'(push(Value)).

eq ==> '$blt_st'(pop(A)), '$blt'(eq(A,B)), '$blt_st'(push(B)).

compare(Ty) ==> '$blt_st'(pop(A)), '$blt_st'(pop(B)),
    '$blt'(~compare(Ty,A,B,C)),
    '$blt_st'(push(C)).
compare(Type,A,B,C) := Type = string ? compare_string(A,B,C)
    | Type = pair(Tl,Tr) ? compare_pair(Tl,Tr,A,B,C)
```

```
33      | Type = int ? compare_int(A,B,C)
34      | Type = nat ? compare_int(A,B,C)
35      | Type = timestamp ? compare_timestamp(A,B,C)
36      | Type = mutez ? compare_tez(A,B,C)
37      | Type = bool ? compare_bool(A,B,C)
38      | Type = address ? compare_address(A,B,C)
39      | Type = key_hash ? compare_key_hash(A,B,C).
40
41  add_intint ==> '$blt_st'(pop(X)), '$blt_st'(pop(Y)),
        '$blt'(add(X,Y,Z)), '$blt_st'(push(Z)).
42
43  mul_intint ==> '$blt_st'(pop(X)), '$blt_st'(pop(Y)),
        '$blt'(mul(X,Y,Z)), '$blt_st'(push(Z)).
44
45  cons_pair ==> '$blt_spec'(cons_pair,2,1,'$cons_pair').
46
47  car ==> '$blt_spec'(car,1,1,'$car').
48
49  cdr ==> '$blt_spec'(cdr,1,1,'$cdr').
50
51  list_size ==> '$blt_st'(pop(A)), '$blt'(list_size(A,B)),
        '$blt_st'(push(B)).
52
53  list_iter(Body) ==> '$blt_st'(pop(List)), list_iter(List,Body).
54  list_iter([],_) ==> '$blt'('$list_iter_end').
55  list_iter([X|Xs],Body) ==> '$blt'('$list_iter'),
56      '$blt_st'(push(X)), '$run'(Body), list_iter(Xs,Body).
57
58  nil ==> '$blt'(nil(L)), '$blt_st'(push(L)).
59
60  if_cons(Bt,Bf) ==> '$blt_st'(pop(Xs)), '$blt'('$if_cons'),
        if_cons_(Xs,Bt,Bf).
61  if_cons_([],_,Bf) ==> '$run'(Bf).
62  if_cons_([X|Xs],Bt,_) ==>
63      '$blt_st'(push(Xs)), '$blt_st'(push(X)), '$run'(Bt).
```

Listing B.1: Definition of the instructions used in the example translation.

Appendix C

Cost model

```
:- package(_).

:- use_package(assertions).
:- use_package(resdefs).

:- use_module(ciao_tezos(michelson_preds)).
:- use_module(ciao_tezos(michelson_types)).

:- resource michelson_gas.

:- compound_resource(michelson_gas, exp(2,-7) * (
    2 * michelson_allocations
    + michelson_steps
    + michelson_reads * 100
    + michelson_writes * 160
    + michelson_bytes_read * 10
    + michelson_bytes_written * 15
)).

:- resource michelson_allocations.
:- resource michelson_steps.
:- resource michelson_reads.
:- resource michelson_writes.
:- resource michelson_bytes_read.
:- resource michelson_bytes_written.

:- default_cost(ub,michelson_steps,0).
:- default_cost(lb,michelson_steps,0).
:- head_cost(ub, michelson_steps, 0).
:- head_cost(lb, michelson_steps, 0).
:- literal_cost(ub, michelson_steps, 0).
:- literal_cost(lb, michelson_steps, 0).
:- trust_default + cost(ub, michelson_steps, 0).
:- trust_default + cost(lb, michelson_steps, 0).

```

```
:- default_cost(ub,michelson_allocations,0).
:- default_cost(lb,michelson_allocations,0).
:- head_cost(ub, michelson_allocations, 0).
:- head_cost(lb, michelson_allocations, 0).
:- literal_cost(ub, michelson_allocations, 0).
:- literal_cost(lb, michelson_allocations, 0).
:- trust_default + cost(ub, michelson_allocations, 0).
:- trust_default + cost(lb, michelson_allocations, 0).

:- head_cost(ub, michelson_reads, 0).
:- head_cost(lb, michelson_reads, 0).
:- literal_cost(ub, michelson_reads, 0).
:- literal_cost(lb, michelson_reads, 0).
:- trust_default + cost(ub, michelson_reads, 0).
:- trust_default + cost(lb, michelson_reads, 0).

:- head_cost(ub, michelson_writes, 0).
:- head_cost(lb, michelson_writes, 0).
:- literal_cost(ub, michelson_writes, 0).
:- literal_cost(lb, michelson_writes, 0).
:- trust_default + cost(ub, michelson_writes, 0).
:- trust_default + cost(lb, michelson_writes, 0).

:- head_cost(ub, michelson_bytes_read, 0).
:- head_cost(lb, michelson_bytes_read, 0).
:- literal_cost(ub, michelson_bytes_read, 0).
:- literal_cost(lb, michelson_bytes_read, 0).
:- trust_default + cost(ub, michelson_bytes_read, 0).
:- trust_default + cost(lb, michelson_bytes_read, 0).

:- head_cost(ub, michelson_bytes_written, 0).
:- head_cost(lb, michelson_bytes_written, 0).
:- literal_cost(ub, michelson_bytes_written, 0).
:- literal_cost(lb, michelson_bytes_written, 0).
:- trust_default + cost(ub, michelson_bytes_written, 0).
:- trust_default + cost(lb, michelson_bytes_written, 0).

:- trust pred '$cdr'
   + ( not_fails, is_det, cardinality(1,1),
       cost(ub, michelson_steps, 20), cost(lb, michelson_steps, 20)
     ).

:- trust pred '$car'
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_steps,20), cost(ub,michelson_steps,20)
     ).

:- trust pred nil(L)
   => ( list(L), gnd(L), size(ub,L,0), size(lb,L,0) )
```

Cost model

```
85    + ( not_fails, is_det, cardinality(1,1),
86        cost(ub, michelson_steps, 20), cost(lb, michelson_steps, 20)
87      ).
88
89  :- trust pred '$cons_pair'
90    + ( not_fails, is_det, cardinality(1,1),
91        cost(ub, michelson_steps, 20), cost(lb, michelson_steps, 20)
92      ).
93
94  :- trust pred '$dup'
95    + ( not_fails, covered, is_det, cardinality(1,1),
96        cost(lb,michelson_steps,20), cost(ub,michelson_steps,20)
97      ).
98
99  :- trust pred '$dip'
100   + ( not_fails, covered, is_det, cardinality(1,1),
101       cost(lb,michelson_steps,40), cost(ub,michelson_steps,40)
102     ).
103
104 :- trust pred '$drop'
105   + ( not_fails, covered, is_det, cardinality(1,1),
106       cost(lb,michelson_steps,20), cost(ub,michelson_steps,20)
107     ).
108
109 :- trust pred '$const'(A)
110   => gnd(A)
111   + ( not_fails, covered, is_det, cardinality(1,1),
112       cost(lb,michelson_steps,20), cost(ub,michelson_steps,20)
113     ).
114
115 :- trust pred add(A,B,C)
116   => ( int(A),
117        int(B),
118        int(C), size(ub,C,int(A)+int(B)), size(lb,C,int(A)+int(B))
119      )
120   + ( not_fails, covered, is_det, cardinality(1,1),
121       cost(lb,michelson_steps,102 + log2(max(int(A),int(B)))/248),
122       cost(ub,michelson_steps,102 + log2(max(int(A),int(B)))/248)
123     ).
124
125 :- trust pred sub(A,B,C)
126   => ( int(A),
127        int(B),
128        int(C), size(ub,C,int(A)-int(B)), size(lb,C,int(A)-int(B))
129      )
130   + ( not_fails, covered, is_det, cardinality(1,1),
131       cost(lb,michelson_steps,102 + log2(max(int(A),int(B)))/248),
132       cost(ub,michelson_steps,102 + log2(max(int(A),int(B)))/248)
133     ).
```

```
134
135 :- trust pred add_tez(A,B,C)
136    => ( mutez(A),
137         mutez(B),
138         mutez(C), size(ub,C,int(A)+int(B)), size(lb,C,int(A)+int(B))
139       )
140    + ( not_fails, covered, is_det, cardinality(1,1),
141        cost(lb,michelson_steps,122),
142        cost(ub,michelson_steps,122)
143      ).
144
145 :- trust pred mul(A,B,C)
146    => ( int(A),
147         int(B),
148         int(C), size(ub,C,int(A)*int(B)), size(lb,C,int(A)*int(B))
149         )
150    + ( not_fails, covered, is_det, cardinality(1,1),
151        cost(lb,michelson_steps,102 +
152            (log2(max(int(A),int(B)))+8)*
153                log2(log2(max(int(A),int(B)))/8+1)
154            / 24
155        ),
156        cost(ub,michelson_steps,102 +
157            (log2(max(int(A),int(B)))+8)*
158                log2(log2(max(int(A),int(B)))/8+1)
159            / 24
160        )
161      ).
162
163 :- trust pred abs(A,B)
164    => ( int(A), int(B), size(ub,B,int(A)), size(lb,B,int(A)) )
165    + ( not_fails, covered, is_det, cardinality(1,1),
166        cost(lb,michelson_steps,122 + (1 + log2(int(A))/8) / 35),
167        cost(ub,michelson_steps,122 + (1 + log2(int(A))/8) / 35)
168      ).
169
170 :- trust pred '$if_none'
171    + ( not_fails, covered, is_det, cardinality(1,1),
172        cost(lb,michelson_steps,40), cost(ub,michelson_steps,40)
173      ).
174
175 :- trust pred cons_none(A)
176    => ( gnd(E) )
177    + ( not_fails, covered, is_det, cardinality(1,1),
178        cost(lb,michelson_steps,20), cost(ub,michelson_steps,20),
179      ).
180
181 :- trust pred amount(A)
182    => ( mutez(A), size(lb,A,int(A)), size(ub,A,int(A)) )
```

```
    + ( not_fails, covered, is_det, cardinality(1,1),
        cost(lb,michelson_steps,20), cost(ub,michelson_steps,20)
      ).

:- trust pred failwith(A,B)
   => ( list(A), is_failed(B) )
   + ( not_fails, covered, is_det, cardinality(1,1) ).

:- trust pred '$swap'
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_steps,20), cost(ub,michelson_steps,20)
     ).

:- trust pred bytes_size(A,B)
   => ( bytes(A), int(B), size(ub,B,length(A)),
      size(lb,B,length(A)) )
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_steps,20), cost(ub,michelson_steps,20)
     ).

:- trust pred slice_bytes(A,B,C,D)
   => ( nat(A), nat(B), bytes(C), gnd(C), option(bytes,D), gnd(D) )
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_steps,80), cost(ub,michelson_steps,80 +
          int(B)/35)
     ).

:- trust pred '$list_map'
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_steps,80), cost(ub,michelson_steps,80) ).

:- trust pred '$list_map_end'
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_steps,60), cost(ub,michelson_steps,60) ).

:- trust pred set_delegate(A,B)
   => ( option(A), operation(B) )
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(ub, michelson_writes, 1), cost(lb,michelson_writes,1),
       cost(ub, michelson_bytes_written, exp(2,7)*32),
       cost(lb, michelson_bytes_written, exp(2,7)*32),
       cost(ub, michelson_steps, exp(2,7) * 10),
       cost(lb, michelson_steps, exp(2,7)*10)
     ).
```

Listing C.1: Cost model including only the instructions used in the examples.

Appendix D

Michelson contract: addition

```
parameter int ;
storage int ;
code { DUP ;
       DIP { CDR @__slash_1 } ;
       DIP { DROP } ;
       CAR @parameter_slash_2 ;
       PUSH int 1 ;
       ADD ;
       NIL operation ;
       NONE key_hash ;
       SET_DELEGATE ;
       CONS ;
       PAIR }
```

Listing D.1: Michelson contract with $O(1)$ complexity and arithmetic operations.

```
:-module(A,[code/3,amount/1],[ciao_tezos(michelson_costs),regtypes]).
:-use_module(ciao_tezos(michelson_preds)).
:-regtype(return_stack/1).
return_stack([(A,B)]) :-
    list(operation,A),
    int(B).
return_stack(A) :-
    is_failed(A).
:-redefining(amount/1).
amount(A).
:-entry(code(A,B,C):(int(A),int(B),var(C))).
code(A,B,[(C,D)]) :-
    '$dup',
    '$dip',
    '$cdr',
    '$dip',
    '$drop',
    '$car',
    '$const'(1),
    add(1,A,D),
```

```
    nil(E),
    cons_none(F),
    set_delegate(F,G),
    cons(G,E,C),
    '$cons_pair'.
```

Listing D.2: Horn Clauses representation of the contract in Listing D.1

```
:- true pred code(A,B,C)
   : ( int(A), int(B), var(C) )
  => ( int(A), int(B), rt4(C),
       size(ub,A,int(A)), size(ub,B,int(B)), size(ub,C,1) )
       + ( cost(ub,michelson_allocations,0),
           cost(ub,michelson_bytes_read,0),
           cost(ub,michelson_bytes_written,4096.0),
           cost(ub,michelson_gas,0.0009765625*log(2,int(A)+1)+494.0859375),
           cost(ub,michelson_reads,0),
           cost(ub,michelson_steps,0.125*log(2,int(A)+1)+1643.0),
           cost(ub,michelson_writes,1) ).
```

Listing D.3: CiaoPP output when analyzing Listing D.2.

Appendix E

Michelson contract: bytes

```
parameter nat ;
storage bytes ;
code { UNPAIR ;
       DIP { DUP ; SIZE ; PUSH int 1 ; SWAP ; SUB ; ABS } ;
       SLICE ;
       ASSERT_SOME ;
       NIL operation ;
       NONE key_hash ;
       SET_DELEGATE ;
       CONS ;
       PAIR }
```

Listing E.1: Michelson contract with $O(1)$ complexity and $O(log(n))$ cost.

```
:-module(A,[code/3,amount/1],[ciao_tezos(michelson_costs),regtypes]).
:-use_module(ciao_tezos(michelson_preds)).
:-regtype(return_stack/1).
return_stack([(A,B)]) :-
    list(operation,A),
    bytes(B).
return_stack(A) :-
    is_failed(A).
:-redefining(amount/1).
amount(A).
:-entry(code(A,B,C):(nat(A),bytes(B),var(C))).
code(A,B,[(C,D)|E]) :-
    '$dup',
    '$car',
    '$dip',
    '$cdr',
    '$dip',
    '$dup',
    bytes_size(B,F),
    '$const'(1),
    '$swap',
    sub(F,1,G),
```

```
    abs(G,H),
    slice_bytes(A,H,B,I),
    '$if_none',
    if_none__0(I,[],[D|E]),
    nil(J),
    cons_none(K),
    set_delegate(K,L),
    cons(L,J,C),
    '$cons_pair'.
if_none__0(none,[],A) :-
    '$const'('()'),
    failwith(['()'],A).
if_none__0(some(A),[],[A]).
```

Listing E.2: Horn Clauses representation of the contract in Listing E.1

```
:- true pred code(A,B,C)
   : ( nat(A), bytes(B), var(C) )
   => ( nat(A), bytes(B), rt9(C),
        size(ub,A,int(A)), size(ub,B,length(B)), size(ub,C,size(C))
           )
   + ( cost(ub,michelson_allocations,0),
       cost(ub,michelson_bytes_read,0),
       cost(ub,michelson_bytes_written,4096.0),
       cost(ub,michelson_gas,0.0009765625*log(2,length(B)+1)+
           0.00002790178571428571*log(2,length(B)-1)+
           496.2892857142857),
       cost(ub,michelson_reads,0),
       cost(ub,michelson_steps,0.125*log(2,length(B)+1)+
           0.0035714285714285713*log(2,length(B)-1)+
          1925.0285714285715),
       cost(ub,michelson_writes,1) ).

:- true pred code(A,B,C)
   : ( nat(A), bytes(B), var(C) )
   => ( nat(A), bytes(B), rt9(C),
        size(ub,A,int(A)), size(ub,B,length(B)), size(ub,C,size(C))
           )
   + ( cost(ub,michelson_allocations,0),
       cost(ub,michelson_bytes_read,0),
       cost(ub,michelson_bytes_written,4096.0),
       cost(ub,michelson_gas,0.0009765625*log(2,length(B)+1)+
           0.00002790178571428571*log(2,length(B)-1)+
           496.4455357142857),
       cost(ub,michelson_reads,0),
       cost(ub,michelson_steps,0.125*log(2,length(B)+1)+
           0.0035714285714285713*log(2,length(B)-1)+
          1945.0285714285715),
       cost(ub,michelson_writes,1) ).

```

Michelson contract: `bytes`

```
:- true pred if_none__0(_A,_B,A)
   : ( none(_A), list(unifier_elem,_B), list_functor(A) )
   => ( none(_A), list(unifier_elem,_B), is_failed(A),
        size(lb,_A,size(_A)), size(lb,_B,length(_B)),
           size(lb,A,size(A)) )
   + ( cost(lb,michelson_allocations,0),
       cost(lb,michelson_bytes_read,0),
       cost(lb,michelson_bytes_written,0),
       cost(lb,michelson_gas,0),
       cost(lb,michelson_reads,0),
       cost(lb,michelson_steps,0),
       cost(lb,michelson_writes,0) ).

:- true pred if_none__0(_A,_B,A)
   : ( none(_A), list(unifier_elem,_B), list_functor(A) )
   => ( none(_A), list(unifier_elem,_B), is_failed(A),
        size(ub,_A,size(_A)), size(ub,_B,length(_B)),
           size(ub,A,size(A)) )
   + ( cost(ub,michelson_allocations,0),
       cost(ub,michelson_bytes_read,0),
       cost(ub,michelson_bytes_written,0),
       cost(ub,michelson_gas,0.15625),
       cost(ub,michelson_reads,0),
       cost(ub,michelson_steps,20),
       cost(ub,michelson_writes,0) ).
```

Listing E.3: CiaoPP output when analyzing Listing E.2.

Appendix F

Michelson contract: apply

```
parameter (pair int (lambda (pair int int) int)) ;
storage (lambda int int) ;
code { CAR ;
       UNPAIR ;
       APPLY ;
       NIL operation ;
       NONE key_hash ;
       SET_DELEGATE ;
       CONS ;
       PAIR }
```

Listing F.1: Michelson contract with $O(1)$ complexity and $O(log(n))$ cost.

```
:-module(A,[code/5,amount/1],[ciao_tezos(michelson_costs),regtypes]).
:-use_module(ciao_tezos(michelson_preds)).
:-regtype(return_stack/1).
return_stack([(A,B)]) :-
    list(operation,A),
    lambda(B).
return_stack(A) :-
    is_failed(A).
:-redefining(amount/1).
amount(A).
:-entry(code(A,B,C,D,E):(int(A),lambda(B),lambda(C),var(D),var(E))).
code(A,B,C,D,'$apply'(int,A,B)) :-
    '$car',
    '$dup',
    '$car',
    '$dip',
    '$cdr',
    '$apply'(int),
    nil(E),
    cons_none(F),
    set_delegate(F,G),
    cons(G,E,D),
    '$cons_pair'.
```

Listing F.2: Horn Clauses representation of the contract in Listing F.1

```
:- true pred code(A,B,C,D,E)
   : ( gnd(A), term(B), term(C), var(D), var(E) )
   => ( gnd(A), term(B), term(C), term(D), term(E),
        size(ub,A,void(A)), size(ub,B,void(B)), size(ub,C,void(C)),
        size(ub,D,void(D)), size(ub,E,void(E)) )
        + ( cost(ub,michelson_allocations,1152.0),
            cost(ub,michelson_bytes_read,0),
            cost(ub,michelson_bytes_written,4096.0),
            cost(ub,michelson_gas,511.8125),
            cost(ub,michelson_reads,0),
            cost(ub,michelson_steps,1608.0),
            cost(ub,michelson_writes,1) ).
```

Listing F.3: CiaoPP output when analyzing Listing F.2.