Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Métodos Formales en Ingeniería Informática

Trabajo Fin de Máster

Improvements to Parametric Cost Analysis and its Application to Smart Contracts

Autor: Víctor Pérez Carrasco
Director y colaboradores:
Manuel Hermenegildo Salinas,
Pedro López García,
José Francisco Morales

Madrid, 8 de agosto de 2021

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

Trabajo Fin de Máster
Máster Universitario en Métodos Formales en Ingeniería Informática

Título: Improvements to Parametric Cost Analysis and its Application to Smart Contracts

8 de agosto de 2021

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

Resumen

La naturaleza de los contratos inteligentes y las plataformas de *blockchain*, donde los programas están replicados a lo largo de todos los nodos y ejecutar un contrato o aumentar su espacio de almacenamiento implica hacerlo en todos los clientes, hace del análisis de recursos un problema relevante. Esto ha conducido al desarrollo de análisis para plataformas y lenguajes específicos. No obstante, la oferta de lenguajes y modelos de coste en estas plataformas es muy amplia, al igual que su mutabilidad en el tiempo, por lo que las soluciones que faciliten el desarrollo y la adaptación de los análisis de coste son atractivas en este contexto. Exploramos la aplicación de una técnica y una herramienta de análisis de coste genéricas a la inferencia estática de cotas del consumo de gas y almacenameinto en contratos inteligentes. Nuestro enfoque se basa en el *Análisis de Coste Paramétrico*, un método que simplifica la implementación de análisis para inferir cotas seguras del consumo de diferentes recursos haciendo uso de distintos modelos de coste. Además, para soportar diferentes lenguajes, realizamos una traducción previa a una representación intermedia basada en cláusulas de Horn. Para demostrar la aplicabilidad de este método, desarrollamos un análisis para la plataforma Tezos y su lenguaje Michelson.

Abstract

The very nature of smart contracts and blockchain platforms, where program execution and storage are replicated across a large number of nodes, makes resource consumption analysis highly relevant. This has led to the development of analyzers for specific platforms and languages. However, blockchain platforms present significant variability in languages and cost models, as well as over time. Approaches that facilitate the quick development and adaptation of cost analyses are thus potentially attractive in this context. We explore the application of a generic approach and tool for cost analysis to the problem of static inference of gas consumption and storage bounds in smart contracts. The approach is based on *Parametric Cost Analysis*, a method that simplifies the implementation of analyzers for inferring safe bounds on different resources and with different resource consumption models. In addition, to support different input languages, the approach also makes use of translation into a Horn clause-based intermediate representation. To assess practicality we develop an analyzer for the Tezos platform and its Michelson language.

Contents

Chapter 1

Introduction

Smart contracts [1] are programs residing in blockchain platforms, whose inherently replicated nature [2, 3] makes cost analysis crucial. Like other elements in the ledgers, these programs and their state are replicated in every node of the blockchain, so every call to a smart contract is executed by every client and an increase in its static state consumes storage space in every node. This fact has led to the implementation of different techniques to limit execution time and static storage size, such as including upper bounds for the consumption of these resources or charging fees. In order to limit execution time, some smart contract platforms introduce the concept of "gas", a virtual resource that reflects the execution time of each instruction. If a transaction exceeds its allowed *gas* consumption, its execution is aborted and its effects, reverted. However, even after failing due to *gas* exhaustion, the transaction is included in the blockchain and the fees are taken. Similar mechanisms are implemented to limit *storage size*. Thus, the cost of a smart contract execution can be expressed in terms of these two resources: *gas* and *storage*.

As a high consumption of these resources may cause extra fees to be taken from smart contract clients—or even that the execution is aborted—, knowing the cost of running a contract beforehand can be useful. Although some smart contract platforms offer simulation mechanisms to dry run contracts in local nodes before performing a transaction, this is not an ideal solution, as it only gives cost data for specific input values, providing no hard guarantees on the costs that may arise when executing the contract with possibly different arguments. Ideally, one would like to be able to obtain instead guaranteed bounds on this cost statically, or at least through a combination of static and dynamic methods.

Static inference of resource consumption of smart contracts may be beneficial not only to clients, but also to smart contract developers themselves. Denial of service (DoS) attacks via uncontrolled resource consumption is a well-known weakness [4] consisting in the allocation of limited resources by an attacker, preventing future valid clients from accessing the software. In the context of smart contracts, this weakness is expressed in the form of the so called DoS with block gas limit [5], which involves increasing the *storage size* of a smart contract whose *gas* consumption depends on that value, e.g., a smart contract which iterates over an array allocated in its static storage. Due to the immutability of these pieces of code, warning developers of a potentially unbounded *gas* consumption before the contract is deployed is crucial, as this weakness can be easily prevented via defensive programming.

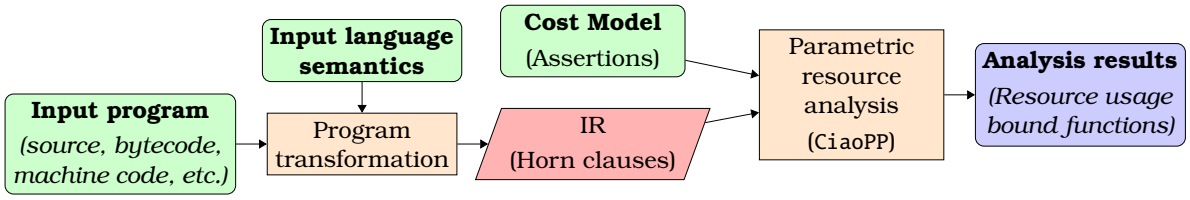

Figure 1.1: Overview of the Parametric Cost Analysis approach.

Thus, formal verification of smart contracts, and in particular analysis and verification of their resource consumption, is receiving increased attention. At the same time, there are currently many blockchain platforms using their own languages and cost models, which often take into account different resources and count them in platform-specific ways, which can also evolve over time. As a consequence, the few existing resource analysis tools for smart contracts, such as GASTAP [6], GASOL [7], or MadMax [8], tend to be quite specific, focusing on just a single platform or language, or on small variations thereof.[1] This makes approaches that would allow quick development of new cost analyses or easily adapting existing ones potentially attractive in this context.

Parametric Cost Analysis is a static analysis technique which allows performing provably correct cost analysis of programs written in different languages and running in their particular platforms [9, 10, 11]. As shown in Figure 1.1, it has been implemented within the `CiaoPP` program development framework [12, 13, 14, 15, 16], which performs combined static and dynamic program analysis, assertion checking, and program transformations, based on computing provably safe approximations of properties, generally using the technique of abstract interpretation [17].

In this approach, language parametricity is achieved via a previous compilation of the source language to a Horn clause-based intermediate representation (CHC IR) [18]. This way, the same analyzers can be used on programs written in languages of variable nature and levels of compilation, ranging from bytecode to higher-level languages such as Java. Regarding platform parametricity, it is achieved thanks to the usage of a rich assertion language in which a vast array of properties of the source program can be expressed via assertions. These assertions form what is called the *cost model*, which transmits the semantics of a language in a given platform to the analyzers by stating useful properties of its atomic elements, e.g., instructions. These properties can be used to state the functional semantics of these elements, e.g., to represent the value of an output w.r.t. the inputs, and they can also provide information about non-functional properties, e.g., execution time or cost [19, 20, 10, 21, 22, 23, 24]. Thus, a cost model can be generated for each blockchain platform and this model can be adapted if the cost of executing the code changes or new resources have to be studied. Given these two elements—the translation to CHC IR and the platform-dependent cost model—we can use the `CiaoPP` framework to infer safe resource usage bound functions depending on the size of input arguments or other parameters for each block in the source program.

A key component of our approach is the aforementioned translation of the target lan-

[1] We discuss this and other relevant related work further in Chapter 2.

```
parameter (list mutez);
storage (list mutez);
code { CAR ; NIL mutez ; SWAP ; ITER { CONS } ; NIL operation ;
    PAIR }
```

Listing 1.1: A Michelson contract that reverses a list.

guages to the CHC IR. A (Constrained) Horn clause ((C)HC) is a (generalized with constraints) first-order logic formula of the form $\forall(S_1 \wedge \cdots \wedge S_n \rightarrow S_0)$ where all the variables in the formula are universally quantified and each S_i is an atomic formula also receiving the name of "literal" or "constraint". These formulae are usually rewritten in the following equivalent form: S_0 `:-` $S_1, \ldots, S_n$, where S_0 is referred to as the *head* of the clause and $S_1, \ldots, S_n$ as its *body*. This translation can be performed by encoding the small-step semantics of the source language [25] or its big-step semantics [18]. In Section 3.1, we will explain how we used the latter approach to implement an automatic Michelson to CHC IR translator.

The Michelson Language is the smart contracts language used in the Tezos platform. It is an interpreted, strongly-typed, and stack-based language which, despite being low-level, provides some high-level abstractions such as data structures, e.g., lists, sets, maps and infinite-precision numbers; or higher-order constructs with instructions like map (`MAP`) or for-each (`ITER`) and anonymous function support (an anonymous function from type `ta` to type `tb` in Michelson belongs to the `(lambda ta tb)` type).

A Michelson contract consists of three sections in arbitrary order: *parameter*, *storage* and *code*. The first two sections state the types of the input argument and the storage respectively, e.g., in Listing 1.1, both are lists of Michelson *mutez* (64-bits unsigned integers), and the code section contains the sequence of Michelson instructions to be executed by the Michelson interpreter. This interpreter can be seen as a pure function from an initial stack to a final stack. The initial stack will contain just a tuple (`Pair Parameter Storage`), whereas the final stack, on success, will consist on just a pair (`Pair Operations Storage`$'$), where `Operations` is the list of blockchain operations to be executed after the contract returns—these operations will be dealt with briefly in a following paragraph—and `Storage`$'$, the updated storage:

$$\begin{aligned} \textit{Interpreter}: \quad & \texttt{pair parameter storage} \ : \ [] \rightarrow \texttt{pair (list operation) storage} \ : \ [] \\ & \texttt{Pair Parameter Storage} \ : \ [] \mapsto \texttt{Pair Operations Storage}' \ : \ [] \end{aligned} \tag{1.1}$$

Michelson instructions can also be seen as pure functions receiving an input stack and returning a result stack. Table 1.1 shows the semantics of instructions in Listing 1.1 in a functional way. Our running example receives an input list and stores the result of reversing it. In order to do so, first, `CAR` discards the previous storage value, as only the parameter list is needed. Then, `NIL mutez` pushes an empty list of mutez in the stack. `SWAP` changes the positions of the top two elements of the stack, obtaining the following stack: `parameter` $:$ $([])$ $:$ $[]$.

Now, we introduce the first higher-order Michelson instruction we discuss in this thesis: `ITER body`. As we can see in Table 1.1, this instruction iterates over the

Table 1.1: Semantics of Michelson instructions in Listing 1.1.

$\texttt{CAR}\colon\ \texttt{pair}\,\texttt{ta}\,\texttt{tb} : \texttt{A} \to\ \texttt{ta} : \texttt{A}$ $\texttt{Pair}\,\texttt{x}\,\texttt{y} : \texttt{S} \mapsto\ \texttt{x} : \texttt{S}$	$\texttt{NIL}\,\texttt{t}\colon\ \texttt{A} \to \texttt{list}\,\texttt{t} : \texttt{A}$ $\texttt{S} \mapsto [] : \texttt{S}$
$\texttt{SWAP}\colon\ \texttt{a} : \texttt{b} : \texttt{A} \to \texttt{b} : \texttt{a} : \texttt{A}$ $\texttt{x} : \texttt{y} : \texttt{S} \mapsto \texttt{y} : \texttt{x} : \texttt{S}$	$\texttt{PAIR}\colon\ \texttt{a} : \texttt{b} : \texttt{A} \to \texttt{pair}\,\texttt{a}\,\texttt{b} : \texttt{A}$ $\texttt{x} : \texttt{y} : \texttt{S} \mapsto \texttt{Pair}\,\texttt{x}\,\texttt{y} : \texttt{S}$
$\texttt{CONS}\colon\ \texttt{ta} : \texttt{list}\,\texttt{ta} : \texttt{A} \to \texttt{list}\,\texttt{ta} : \texttt{A}$ $\texttt{a} : \texttt{l} : \texttt{S} \mapsto (\texttt{a} : \texttt{l}) : \texttt{S}$	
$\texttt{ITER}\,\texttt{body}\colon\ \texttt{list}\,\texttt{t} : \texttt{A} \to \texttt{A}$ $\texttt{body}\colon\ \texttt{t} : \texttt{A} \to \texttt{A}$ $\texttt{l} : \texttt{S} \mapsto \texttt{ITER}(\texttt{l} : \texttt{S}) = \begin{cases} \texttt{S} & \text{if } \texttt{l} = [] \\ \texttt{ITER}(\texttt{l}' : \texttt{body}(\texttt{x} : \texttt{S})) & \text{if } \texttt{l} = \texttt{x} : \texttt{l}' \end{cases}$	

elements of a list, applying the sequence of instructions[2] `body` to each element in the list. In our case, the argument of this instruction is just `{ CONS }`, a singleton sequence containing an instruction which prepends the element on top of the stack to the list right below it. This way, we can define the semantics of this loop as:

$$\texttt{l}_\texttt{a} : \texttt{l}_\texttt{b} : \texttt{S} \mapsto \texttt{ITER}(\texttt{l}_\texttt{a} : \texttt{l}_\texttt{b} : \texttt{S}) = \begin{cases} l_b : S & \text{if } \texttt{l}_\texttt{a} = [] \\ \texttt{ITER}(\texttt{l}'_\texttt{a} : (\texttt{x} : \texttt{l}_\texttt{b}) : \texttt{S}) & \text{if } \texttt{l}_\texttt{a} = \texttt{x} : \texttt{l}'_\texttt{a} \end{cases}$$

Many Michelson instructions receive additional arguments. Other higher-order instructions are `IF` and `LOOP`, which receive one and two blocks of code; some instructions like `NIL` or `SET` receive the type of the data structure to create; `PUSH` receives the type and constant representing the value to introduce in the stack.

Following on with our example, once the input list has been reversed, the `NIL` instructions pushes an empty list of operations in the stack and `PAIR` builds the returning pair from the two elements in the stack, obtaining the desired stack shape:

$$\texttt{pair}\ (\texttt{list operation})\ \texttt{storage} : [], \text{with } \texttt{storage} = (\texttt{list mutez})$$

As an example, a call to this contract with the list of numbers from 1 to 3 as a parameter would present the following input ($\texttt{S}_0$) and output ($\texttt{S}_1$):

$$\texttt{S}_0 = \texttt{Pair}\ [1, 2, 3]\ \_ : [] \mapsto \texttt{S}_1 = \texttt{Pair}\ []\ [3, 2, 1] : []$$

Regarding the *external operations* that Michelson contracts return in the first field of the return tuple, they are the instructions to be performed at the blockchain level. There are three types of operations: *transactions* (operations to transfer tokens and parameters to a smart contract), *originations* (to create new smart contracts given the required arguments), or *delegations* (operations that assign a number of tokens to the stake of another account, without transferring them).

In the rest of this thesis, we start by providing context on the state of the art in Chapter 2, and on the state of our tool before the development of the thesis began in

[2]Michelson also supports macro instructions, which are translated to the sequence of instructions they represent in an early compilation stage.

Chapter 3. In Chapter 4, we show the improvements implemented in the translation tool to better support Michelson features, and, in Chapter 5, we discuss the improvements implemented in the `CiaoPP` tool to handle singularities of Michelson contracts. In Chapter 6, we present some results obtained by analyzing several Michelson contracts using our proposed method. Finally, Chapter 7 presents our conclusions and discusses future work.

Chapter 2

Related Work

As previously stated, tools that have been proposed to date for cost analysis of smart contracts tend to be platform- and language-specific. GASPER [26] and MadMax [8] are both aimed at identifying parts of contracts that have high gas consumption in order to optimize them or to avoid gas-related vulnerabilities—such as the aforementioned DoS with block size limit. While the former recognizes control-flow patterns using symbolic computation the latter searches for both control- and data-flow patterns. Marescotti et al. [27] also use a limited-depth path exploration approach to estimate worst-case gas consumption. These tools are useful programmer aids for preventing attacks from malicious agents in a context in which dependability is crucial, but cannot provide safe cost bounds. GASPER and MadMax are specific to contracts written for the Ethereum platform [2], in Solidity, and translated to Ethereum Virtual Machine (EVM) bytecode. The Solidity compiler provides a platform-specific tool to generate gas constant bounds, i.e., bounds which cannot depend on any input parameters, in which case, the generated bound is infinite.

On the other hand, there are other tools which are closer to our work, such as GASTAP [6] and its extension GASOL [7]. These can infer upper bounds for gas consumption, using similar theoretical concepts as those used by `CiaoPP`, i.e., recurrence relation solving combined with ranking functions, etc. GASOL is a more evolved version of GASTAP that includes optimizations and allows users to choose between a number of predefined configuration options, such as counting particular types of instructions or storage. These are powerful tools that have been proven effective at inferring accurate gas bounds with reasonable analysis times, in a good percentage of cases. However, they are also specific to Ethereum Solidity contracts and EVM.

Parametric Cost Analysis (also referred to as user-defined resource analysis) was proposed in [9] and further developed in [10, 11]. The approach builds on Wegbreit's seminal work [28] and the first full analyzers for upper bounds, in the context of task granularity control in automatic program parallelization [29, 30]. This in turn evolved to cover other types of approximations (e.g., lower bounds [31]), and to the idea of supporting user-defined resources [9, 10]. This analysis was extended to be fully based on abstract interpretation [17] and integrated into the PLAI multi-variant framework, leading to context-sensitive cost analyses [11]. Other extensions include static profiling [32], static bounding of run-time checking overhead [33], or analysis of parallel programs [34]. Other applications include the previously mentioned analyses of platform-dependent properties such as time or energy [19, 20, 10, 21, 22, 23, 24].

Cost analysis has received considerable additional attention lately [35, 36, 37, 38, 39, 40, 41, 42, 36, 43, 44, 45, 46, 47, 48, 49, 50, 51]. While these approaches are not based on the same idea of user-level parametricity that is instrumental in the approach proposed herein, we believe the parametric approach is also relevant for these analyses.

Chapter 3

Previous Work

In this chapter, we will present previous work in this line corresponding to the development of my bachelor thesis [52]. In Section 3.1, we will explain our first approach to the Michelson to CHC IR translation; and in Section 3.2 we will go through the process of obtaining our first Michelson cost model. Together, these elements constitute the basis for a first version of our smart contract analysis tool: the "`ciao_tezos`" tool.

3.1. Michelson to CHC IR Translation: A First Approach

In order to obtain a Michelson to CHC IR compiler, we implemented a Michelson interpreter as a big-step recursive interpreter and obtained instructions semantics as a direct transliteration to CHC in the `Ciao` system [53].

In Table 3.1, we show the CHC encoding of instructions in Listing 1.1. As we can see, there exists a strong correspondence with their semantics, shown in Table 1.1, using Herbrand terms to represent Michelson data structures and lists to represent the stack. Our recursive Michelson interpreter could be encoded as follows:

```
run([], S, S).
run([I|Is], S0, S) :-
    ins(I, S0, S1),
    run(Is, S1, S).

% One clause per I/n instruction:
ins(<<I>>(A1, ..., An), S0, S) :-
    <<I>>(A1, ..., An, S0, S1)
```

The recursive predicate `run/3` receives a list of instructions to execute and the initial stack and returns the result stack after running such instructions in its third argument. In order to do so, it makes use of the dispatcher (`ins/3`), which runs the definition of the current instruction (as those found in Table 3.1). Based on this, we derived a simple translator based on a specialization of a CHC partial evaluation algorithm for this particular recursive interpreter.

As this translator implemented some simple partial evaluation rules to help produce friendlier code for the analyzer, in some cases the resulting code would omit some

Table 3.1: CHC representation of instructions in Listing 1.1.

```
car([(X,_)|S], [X|S]).     nil(S, [[]|S]).

swap([X,Y|S], [Y,X|S]).    pair([X,Y|S], [(X,Y)|S]).

            cons([A,L|S], [[A|L]|S]).

            iter(Body, [L|S0], S1) :-
                iter(L, Body, S0, S1).

            iter([], _, S, S).
            iter([X|Xs], Body, S0, S2) :-
                run(Body, [X|S0], S1),
                iter(Xs, Body, S1, S2).
```

instructions which simply modified the stack and simple data structures inside it. In order to prevent this, we included *cost markers* in their definition, no-ops which could be used to preserve the cost semantics of the resulting code.

Thanks to Michelson typing rules, we are able to infer the type of the stack, i.e., its length and the type of its elements, at each program point. With this information, we can specialize polymorphic instructions, depending on the type of the input. This is a big step forwards, as instructions semantics and *cost semantics* may depend on the type of the arguments of each instruction. This way, e.g., we translate the `ADD` instruction to any of:

$$\texttt{ADD}[\texttt{A},\texttt{B}] \rightarrow \begin{cases} \texttt{add\_intint} & \text{if } \texttt{int}(\texttt{A}), \texttt{int}(\texttt{B}) \\ \texttt{add\_intnat} & \text{if } \texttt{int}(\texttt{A}), \texttt{nat}(\texttt{B}) \\ \texttt{add\_natint} & \text{if } \texttt{nat}(\texttt{A}), \texttt{int}(\texttt{B}) \\ \texttt{add\_natnat} & \text{if } \texttt{nat}(\texttt{A}), \texttt{nat}(\texttt{B}) \\ \texttt{add\_timestamp\_to\_seconds} & \text{if } \texttt{timestamp}(\texttt{A}), \texttt{int}(\texttt{B}) \\ \texttt{add\_seconds\_to\_timestamp} & \text{if } \texttt{int}(\texttt{A}), \texttt{timestamp}(\texttt{B}) \\ \texttt{add\_tez} & \text{if } \texttt{mutez}(\texttt{A}), \texttt{mutez}(\texttt{B}) \end{cases} \tag{3.1}$$

Regarding higher-order instructions, e.g., `MAP body`, `IF bt bf`, etc., our definitions of these instructions receive three parameters: the control condition, the input stack and the output stack. Thus, e.g., the `ITER body` instruction is defined as seen in Table 3.1.

Using the aforementioned interpreter and instructions annotations, we generated a predicate `«I»__«index»/3` for each instantiation of these instructions found in the original code, e.g., an instruction `ITER { CONS }` would translate to—assuming an index of 1:

```
iter__1([], S, S).
iter__1([X|L], S0, S) :-
    run([cons], [X|S0], S1),
    iter__1(L, S1, S).
```

Through simple partial interpretation, we obtained:

```
iter__1([], S, S).
iter__1([X|L0], [L1|S0], S) :-
    cons(X, L1, L2),
    iter__1(L0, [L2|S0], S).
```

The main code section of a contract was treated in a similar way, generating a single-clause predicate `code/3`, receiving as arguments the parameter, the storage and the output stack. This way, our running example (Listing 1.1) was translated to something similar to:

```
:- entry code(P, S, Res)
    : ( list(mutez, P), list(mutez, S), var(Res) ).

code(P, _, [([],R)]) :-
    '$car', nil([]), '$swap', '$iter',
    iter__0(P, [[]], [R]),
    nil([]), '$cons_pair'.

iter__0([], S, S).
iter__0([X|L], [R0|S0], S) :-
    cons(X, R0, R1),
    iter__0(L, [R1|S0], S).
```

3.2. Our First Michelson Cost Model

In this section, we will cover the procedure we followed in order to obtain a precise and correct cost model for Tezos' Michelson contracts previously translated to CHC IR.

As we have seen, a cost model is a definition of the semantics and cost 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 cost semantics of the platform used, the more precise the analysis will be.

The process of obtaining a cost model may involve applying profiling techniques to measure the amount of each resource, e.g., energy or time, consumed by the built-in instructions. In this case, we are studying virtual resources, so there was no need to measure the performance of real machines running the code. Instead, we obtained our cost model by extracting it from the platform's source code hosted on GitLab. This method consisted in inspecting hundreds of lines of code written in Ocaml, so it also involved learning a new language. Tezos releases a new protocol version twice a year, and each new version changes the way gas is counted. Our first cost model targeted Tezos' *Carthage* protocol.

By inspecting the source code, we collected valuable information about the Tezos platform. Regarding gas, we learned that it is not an atomic resource, but a compound one, i.e., it can be expressed in terms of other resources. For the Carthage cost model:

```
:- 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.

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

Listing 3.2: Assertions to declare *gas* as a compound resource in Carthage cost model.

$$gas(allocations, steps, reads, writes, bytes\_read, bytes\_written) = \\ = 2^{-7} * \begin{pmatrix} allocations \\ steps \\ reads \\ writes \\ bytes\_read \\ bytes\_written \end{pmatrix} \times \begin{pmatrix} 2 \\ 1 \\ 100 \\ 160 \\ 10 \\ 15 \end{pmatrix} \tag{3.2}$$

In our cost model we first named the resources as in Listing 3.1 and then defined `michelson_gas` as a compound resource following Equation 3.2 (Listing 3.2).

Each Michelson instruction will consume one or more of these basic resources, which will later be used to calculate *gas* consumption. Once the resources to be inferred by the analysis have been included in the cost model, we can proceed to also declare this consumption. Since in most cases not all resources will be consumed by every instruction, we include in the model some default cost assertions establishing, for example, that the consumption of these basic resources is 0 by default (Listing 3.3).

```
:- 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 3.3: Assertions to declare the default cost of a resource.

```
:- trust pred add_intint(A,B,C)
   => ( int(A), int(B), int(C),
        size(ub,C,int(A)+int(B)),
        size(lb,C,int(A)+int(B)) )
   + ( not_fails, is_det, cardinality(1,1),
       cost(lb,michelson_steps,
            102+(1+log2(max(int(A),int(B)))/8)/31),
       cost(ub,michelson_steps,
            102+(1+log2(max(int(A),int(B)))/8)/31))
```

Listing 3.4: Cost assertion for `add_intint` in the Carthage cost model.

In Listing 3.4, we can see an example of an assertion included in our first cost model. In this case, we state that the cost for the `ADD` instruction when both operands are integers (which is internally represented as `add_intint` after specializing the instruction) is logarithmic with respect to the maximum of both operands. This assertion not only includes cost-related information, but also typing and *size* information, which is crucial for cost analysis.

As seen in Listing 3.4, cost and size related *properties* must state what *bound* they refer to. In this case, both upper and lower bounds are given. As they are the same, we know the cost is exact—these two properties can be expressed with a single property using the keyword `exact` instead.

Chapter 4

`ciao_tezos` Improvements

In this chapter, we will focus on the improvements the `ciao_tezos` tool has gone through during the development of this Masters thesis. Firstly, in Section 4.1, we will go through the process of supporting a new Tezos protocol. Then, in Section 4.3, we will discuss the improvements we introduced to Tezos cost models. Section 4.2 shows improvements to the translation process in order to obtain a friendlier CHC IR, also explaining its motivation by giving an insight to `CiaoPP`'s size analysis. Section 4.4 covers slight improvements to Michelson partial evaluation. And, last but not least, Sections 4.5 and 4.6 introduce the reader to a new Michelson concept, *entrypoints*, and show how defining a native Michelson assertion language to write `Ciao` assertions in smart contracts can be useful to Michelson developers.

4.1. Supporting New Tezos Protocols

Since the presentation of my Bachelor thesis, Tezos has released two new protocol versions: *Delphi* and *Edo*. In this section, we will go through the changes that we introduced in order to support these updates to Tezos protocol.

In the case of Delphi, the Michelson language remained unchanged, so we simply had to define a new cost model to reflect the modifications in Tezos cost semantics. We could achieve this in under a day, writing around 500 lines of code. We would like to point out that this fact clearly proves the feasibility and usefulness of Parametric Cost Analysis in a dynamic context such as smart contract programming languages. The main change introduced in this update was the inclusion of a new atomic resource influencing gas consumption, *atomic_steps*. In order to support this, we simply had to modify our gas definition, as can be seen in Listing 4.1. In Listing 4.2, we show the cost assertion defining the cost for the `ADD` operation when both operands are integers (as in Listing 3.2).

With respect to the Edo protocol, this update did not bring as many adjustments to the cost model as Delphi, but it did introduce some new features to the Michelson language, e.g., a new `never` type, new instructions, tuples (implemented as recursive pairs) and new cryptographic capabilities. In order to support these new constructs, we had to modify our Michelson to CHC IR translator and include their semantics in the updated cost model.

```
:- compound_resource(michelson_gas, atomic_steps + 1000 * (
    michelson_allocations * 2
    + michelson_steps
    + michelson_reads * 100
    + michelson_writes * 160
    + michelson_bytes_read * 10
    + michelson_bytes_written * 15
)).
```

Listing 4.1: Assertions to declare *gas* as a compound resource in Delphi cost model.

```
:- trust pred add_intint(A,B,C)
   => ( int(A), int(B), int(C),
        size(ub,C,int(A)+int(B)),
        size(lb,C,int(A)+int(B)) )
   + ( not_fails, covered, is_det, cardinality(1,1),
       cost(lb,michelson_atomic_steps,80 +
            (1 + log2(max(int(A),int(B)))/8) * 2 ** -4 +
            (1 + log2(max(int(A),int(B)))/8) * 2 ** -6),
       cost(lb,michelson_atomic_steps,80 +
            (1 + log2(max(int(A),int(B)))/8) * 2 ** -4 +
            (1 + log2(max(int(A),int(B)))/8) * 2 ** -6) ).
```

Listing 4.2: Cost assertion for `add_intint` in the Delphi cost model.

We provide a simple mechanism based on configuration flags to configure the Michelson to CHC IR translation. Through these flags, the user can also select the target Tezos protocol. In most cases—except when the contract contains new or no longer supported instructions—the translation result will be identical for every protocol. The main difference will reside in the `Ciao` module definition. Tezos cost models are implemented as `Ciao` packages and, by loading different packages, we can easily select what cost model we want to use. As we can see in Listing 4.3, by simply changing the `module` definition in the translated module, we can analyze the same contract using different cost models.[1]

4.2. An Analysis-Friendlier Translation

Throughout this section, we will cover the updates that the translation tool has suffered in order to make its output easier to analyze. In Section 4.2.1, we will introduce the size analysis used, whose limitations motivate the changes made to the tool. Then, in Sections 4.2.2 and 4.2.3, we will cover two important modifications made to

[1] We will show results for different cost models in Chapter 6.

```
:- module(_, [], [ciao_tezos(cost_models/carthage)]).
...
:- module(_, [], [ciao_tezos(cost_models/delphi)]).
...
:- module(_, [], [ciao_tezos(cost_models/edo)]).
```

Listing 4.3: Different module definitions for each cost model.

```
:- entry default(A, B) : ( list(gnd, A), var(B) ).

default([(Parameter,Storage0)], [(Operations,Storage1)]) :-
    ...
```

Listing 4.4: A primitive CHC IR representation of the contract in Listing 1.1.

our translation tool: *tuple destructuring* and *stack deforestation*.

4.2.1. An Introduction to Size Analysis

In this section, we will introduce the size analysis used by CiaoPP's cost analysis [54]. This is a relatively old size analysis, whose limitations are addressed in newer domains such as, e.g., the *sized types* domain [55].

Our size analysis infers a set of argument size relations for each clause in order to represent the upper and lower bound of the input size to each body literal as a function in terms of the input size to the head. This size information is used by the cost analysis to infer the cost of each literal in the body of a clause.

Each literal can be expressed in terms of four measures: *term size*, *term depth*, *list length* or *integer value*. In order to decide which of these measures to apply, type information from an instrumental type analysis is used. Thus, we can define a set of functions $|\cdot|_m : \mathcal{H} \to \mathbb{N}_\perp$, $\mathcal{H}$ being the Herbrand universe and $\mathbb{N}_\perp$ the set of natural numbers with an additional symbol: $\perp$, representing an undefined size:

$$size_{int}(t) = \begin{cases} n & \text{if } t \text{ is an integer } n \\ \odot(size_{int}(t_1), \ldots, size_{int}(t_n)) & \text{if } t = \odot(t_1, \ldots, t_n), \odot \text{ is an arithmetic functor} \\ \perp & \text{otherwise} \end{cases}$$

$$size_{list}(t) = \begin{cases} 0 & \text{if } t = [] \\ 1 + size_{list}(t_1) & \text{if } t = [\_|t_1] \\ \perp & \text{otherwise} \end{cases}$$

$$size_{depth}(t) = \begin{cases} 0 & \text{if } t \text{ is a constant} \\ 1 + \max\{size_{depth}(t_i) \mid 1 \leq i \leq n\} & \text{if } t = f(t_1, \ldots, t_n) \\ \perp & \text{otherwise} \end{cases}$$

$$size_{size}(t) = \begin{cases} 1 & \text{if } t \text{ is a constant} \\ 1 + \sum_{i=1}^{n} size_{size}(t_i) & \text{if } t = f(t_1, \ldots, t_n) \\ \perp & \text{otherwise} \end{cases}$$

As we can see, these measures are quite limited for analyzing term sizes in a language as expressive as our CHC IR. E.g., if we directly translate our running example (Listing 1.1), we would get a predicate like that in Listing 4.4.

This program, however, would be almost impossible to analyze using the previously defined measures. Although the input terms are `Parameter` and `Storage`, we cannot use their sizes to express the size of the body literals in the clause, as we can only use the size of the only input argument: the input stack (`[(Parameter,Storage)]`), whose

```
:- entry default(A, B, C)
    : ( list(mutez, A), list(mutez, B), var(C) ).

default(Parameter, Storage0, [(Operations,Storage1)]) :-
   ...
```

Listing 4.5: Second iteration on the translation output (see Listing 4.4).

```
parameter (pair (pair int int) int) ;
storage (pair int int) ;
code { ... }
```

Listing 4.6: Michelson contract with tuples.

size is always 1 using the measure *list_length*—measures *term_size* and *term_depth* are applicable to any term, but we will only use *list_length* for lists. Following this reasoning, during the development of my Bachelor thesis, we modified our translator to be able to use the size of these terms in the analysis, such as in Listing 4.5.

However, this change, although useful, is not enough. Looking at the contract in Listing 4.6, we can see that, even after applying this transformation to the output of the translator, the size of the input arguments is not accessible for our analysis.

By the output in Listing 4.7, we can infer that our size analysis can only use the measures *term_size* and *term_depth* (for constant sizes of 5 and 3, respectively). Our first intuition is that using the sizes of the terms in the input tuples with the measure *integer_value* would be much more useful. To this end, in Section 4.2.2, we introduce a fix to this problem: *tuples destructuring*.

Another issue with the current representation of Michelson smart contracts is that we cannot express the sizes of the outputs, i.e., the list of operations and the updated storage, in terms of the inputs, as there is a single output argument in our CHC IR representation using the *list_length* measure. In Listing 3.1, we can also see how our representation of the stack as a list also limits the precision of our analysis, as the auxiliary predicate `iter__1/3`, which simulates the loop in our running example, receives the input and output stacks as a parameter, losing any size information of the terms in the body literals occurring after the call to this predicate. In order to tackle this problem, in Section 4.2.3, we introduce a new concept: *stack deforestation*.

4.2.2. Destructuring Michelson Tuples

Tuple destructuring is a transformation which can be applied to tuples in order to obtain their inner terms statically. Being $\mathcal{T}$ the set of Michelson types, $\mathcal{T}_{dest}$ the set of Michelson types excluding tuples, and $\times$ the associative tuples constructor, we can

```
:- entry default(A, B, C)
    : ( pair(pair(int, int), int, A), pair(int, int, B) var(C) ).

default(((A,B),C), (D,E), [(Operations,Storage1)]) :-
   ...
```

Listing 4.7: Direct translation of the contract in Listing 4.6.

```
:- entry default(A, B, C, D, E, F)
    : ( int(A), int(B), int(C), int(D), int(E), var(F) ).

default(A, B, C, D, E, [(Operations,Storage1)]) :-
   ...
```

Listing 4.8: New translation of the contract in Listing 4.6 using $Interpreter_{dest,\texttt{pair (pair int int) int,pair int int}}$.

define the $dest$ function recursively as follows:

$$dest\colon \mathcal{T} \to \mathcal{T}^n_{dest}, n > 0$$
$$\texttt{a} \mapsto Destructure(\texttt{a}) = \begin{cases} Destructure(\texttt{tl}) \times Destructure(\texttt{tr}) & \text{if } \texttt{a} = \texttt{pair tl tr} \\ \texttt{a} & \text{otherwise} \end{cases}$$

This way, from an arbitrary Michelson tuple type `pair tl tr`, we can obtain a tuple of Michelson types t. Note that the same approach can be applied to Michelson terms in the stack to obtain a tuple of Michelson terms. Being $\mathcal{M}$ the universe of Michelson terms and $\mathcal{M}_{dest}$ this universe without Michelson tuples, we can overload the $dest$ function:

$$dest\colon \mathcal{M} \to \mathcal{M}^n_{dest}, n > 0$$

Going back to Equation 1.1, we can see how we defined a Michelson interpreter as a function from input to output stack. Using this same principle, we can derive a new interpreter, which receives a destructured tuple as a parameter:

$$\begin{aligned} Interp_{dest,\texttt{p,s}}\colon\ & \texttt{p}_1 : \ldots : \texttt{p}_n : \texttt{s}_1 : \ldots : \texttt{s}_m : [] \mapsto \texttt{list operation} : \texttt{s}'_1 : \ldots : \texttt{s}'_m : [] \\ & \texttt{P}_1 : \ldots : \texttt{P}_n : \texttt{S}_1 : \ldots : \texttt{S}_m : [] \mapsto \texttt{Operations} : \texttt{S}'_1 : \ldots : \texttt{S}'_m : [] \end{aligned} \tag{4.1}$$

Note that $\texttt{p}_1, \ldots, \texttt{p}_n, \texttt{s}_1, \ldots, \texttt{s}_m / \texttt{P}_1, \ldots, \texttt{P}_n, \texttt{S}_1, \ldots, \texttt{S}_m$, is the result of applying the $dest$ function to the input type (`pair parameter storage`)/input term (`Pair Parameter Storage`). In order to be able to reuse instructions definitions, this interpreter must "rebuild" input Michelson tuples before executing the contract using type information provided by `p` and `s`, as our only goal is to define an equivalent interpreter receiving a destructured tuple instead of a Michelson tuple.

Using this simple transformation, contracts like that in Listing 4.6 can be represented in an analysis-friendlier way, as seen in Listing 4.8. Using this new representation, we can use the size of each input term inside Michelson tuples to infer the cost of executing a contract, greatly improving the precision of the analysis in the general case.

4.2.3. Stack Deforestation

As seen in Section 4.2.1, passing the stack as a parameter to auxiliary predicates which encode higher-order Michelson instructions can be very detrimental to size analysis precision. In order to avoid that, we can use *stack deforestation*, abstracting the underlying stack in the translation process.

In Section 3.1, we explain how we can use Michelson typing rules (the type of a stack in every program point can be known statically) to specialize polymorphic instructions. This same approach can be applied to abstract the stack, exploiting higher-order instructions typing rules. In order to prove the correctness of our approach, we will go through two key properties that both *branch* and *loop* instructions must satisfy.

Rule 1 (Branch merging rule) *When a program branches, either in a branch or a loop instruction—the loop can be entered or not—, both branches must return the same stack type. In the case of branch instructions, one of them or both of them may fail, if only one fails, the non-failing return type is taken as the result of executing the higher-order instruction.*

Rule 2 (Loop type-preserving rule) *The type of a stack at the exit of a loop must be the same as in its entry plus a* looping term*, be it a condition term, e.g., a* `bool`*, or a new element to insert into the new data structure in* `MAP` *instructions.*[2]

Using these typing rules, we can assure that the size of a stack is defined in each program point, regardless of the number of times it is executed or the instructions it has previously executed. Thus, we can perform stack deforestation to abstract the Michelson stack in every program point, including higher-order instructions, by the following rule:

$$\texttt{ins}(\texttt{Cond}, \texttt{S0}, \texttt{S1}) \mapsto \texttt{ins}(\texttt{Cond}, \texttt{A}_1, \ldots, \texttt{A}_n, \texttt{B}_1, \ldots, \texttt{B}_m), n = \texttt{size}(S0), m = \texttt{size}(S1) \tag{4.2}$$

Using this transformation, our running example (Listing 3.1) can be modified to obtain an analysis-friendlier version:

```
:- entry default(P, S, Ops, Res)
    : ( list(mutez, P), list(mutez, S), var(Ops), var(Res),
        mshare([[Ops],[Res]]) ).

default(A, B, [], C) :-
    '$car', nil([]), '$swap', '$iter',
    iter__1(A, [], C),
    nil([]), '$cons_pair'.

iter__0([], A, A).
iter__0([A|B], C, D) :-
    cons(A, C, [A|C]),
    iter__0(B, [A|C], D).
```

In case one of the branches of an instruction fails, we apply the transformation in Equation 4.2 taking the size of the stack at the exit of the the non-failing branch as the stack size. E.g., in Listing 4.9, we have an `IF` instruction in which one of the branches fails. As the size of the returning stack is 0 in the non-failing case, no output stack is encoded in the head of the generated predicate, as seen in Listing 4.10.

In this previous case, the size of the non-failing stack was lesser than that of the failing branch, but the opposite is also possible. We could run into a program for which our translator generates the following predicate:

[2]This rule follows from the previous one.

```
parameter bool ;
storage unit ;
code { CAR ; IF { FAIL } {} ; UNIT ; NIL operation ; PAIR }
```

Listing 4.9: Michelson contract with a failing branch.

```
default(A,'()',[],'()') :-
    '$car', '$if',
    if__0(A),
    '$const'('()'), nil([]), '$cons_pair'.

if__0(true) :-
    '$const'('()'), failwith('()').
if__0(false).
```

Listing 4.10: CHC IR representation of Listing 4.9.

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

This predicate cannot be analyzed as it is, as the size analysis would lose precision due to the inclusion of the `failed('()')` term and several free variables as output arguments in the head of the predicate. In Section 5.2, we dive into this concrete problem and discuss a possible solution.

4.3. Improvements to Cost Models

In this section we will show how we added support for a new Michelson resource (storage) by slightly modifying our CHC IR and cost model (Subsection 4.3.1). Also, in Subsection 4.3.2, we will discuss how we modified our cost models, detaching semantics from cost semantics.

4.3.1. Storage Resource Support

As we stated in the introduction, most smart contract platforms charge users for increasing the *storage size* of a contract, so knowing how this will change in terms of the inputs can be very useful for smart contract clients.

As we know, our CHC IR translation of Michelson smart contracts encodes the code section of a contract as a single-clause predicate. In Listing 4.11, we can see how inputs and outputs are encoded as terms in the head of the predicate and how, for our running example, the translator generates a predicate `default/4` in which the first two arguments are the input parameter and storage, respectively, and the last two arguments, the output operations list and storage.

Thanks to the transformation applied to input and output stacks explained in Section 4.2, we can think think of the size of a storage term as a vector $[size_0, \ldots, size_{n-1}]$,

```
:- entry default(A, B, C, D)
   : ( list(mutez, A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([[C],[D]])).

default(A, B, [], C) :-
```

Listing 4.11: Head and input types definition of the predicate encoding the code section in Listing 1.1.

```
:- true pred default(A,B,C,D)
   : ( list(int,A), list(int,B), var(C), var(D) )
   => ( list(,A), list(int,B), list(operations,C),
      list(memo_size,D),
        size(ub,length,C,0),
        size(ub,length,D,length(A)) ).
```

Listing 4.12: Size analysis output for the storage parameter in Listing 4.11.

where each $size_i$ represents the size of the storage element i obtained as explained in Section 4.2.2 (if storage is not a tuple, $n = 1$). This way, for any input ($input\_storage$) and output ($output\_storage$) storage vectors of the same length, we have:

$$\Delta_i = output\_storage_i - input\_storage_i \tag{4.3}$$

By performing size analysis on the terms in the head of the predicate, we can obtain the expression that relates the output storage with the input arguments (Listing 4.12). Now, as the output storage can be expressed in terms of the input storage and the parameter, we can modify Equation 4.3, obtaining an expression depending on the inputs to the contract:

$$\Delta_i(parameter, input\_storage) = output\_storage_i(parameter, input\_storage) - input\_storage_i \tag{4.4}$$

However, this expression only gives us the size difference in the storage in terms of parameter and input storage measured in one of the metrics explained in Section 4.2.1, the actual storage difference depends on the encoding of the different types used in Tezos Michelson interpreter. In order to obtain the actual storage difference, we relate this expression with the type of the term measured:

$$michelson\_storage_i(type, parameter, storage) = michelson\_storage_{type_i}(\Delta_i(parameter, storage)) \tag{4.5}$$

This way we obtain an expression from the input of the contracts to the storage difference due to every term in the storage tuple. Thanks to Michelson encoding of tuples, we can easily obtain an expression from input arguments to storage difference:

$$\begin{aligned} &michelson\_storage(type, parameter, storage) = \\ &\sum_{i=0}^{n-1} michelson\_storage_i(type, parameter, storage) \end{aligned} \tag{4.6}$$

As `michelson_storage` is a new resource in our cost model, we must declare it (Listing 4.13). In the Tezos platform, storage is a special resource, as Michelson instructions do not consume it; it has to be measured after executing a contract, once the

```
:- resource michelson_storage.
```

Listing 4.13: `michelson_storage` resource definition in our model.

```
:- trust pred '$storage_diff_bool'(A,B)
   + ( cost(lb,michelson_storage,0),
       cost(ub,michelson_storage,0) ).

:- trust pred '$storage_diff_string'(A,B)
   + ( cost(lb,michelson_storage,length(B)-length(A)),
       cost(ub,michelson_storage,length(B)-length(A)) ).

:- trust pred '$storage_diff_int'(A,B)
   + ( cost(lb,michelson_storage,log(256,int(B))-log(256,int(A))),
       cost(ub,michelson_storage,log(256,int(B))-log(256,int(A))) ).
```

Listing 4.14: Assertions for some `michelson_storage` cost markers.

output storage has been calculated. As a result, assertions in our model declaring gas consumption for each Michelson instruction will not be modified.

However, we have to find a mechanism to measure this new resource. In order to do so, we defined a series of cost markers that are appended as the last instructions of every Michelson contract. These cost markers are obtained from a polymorphic meta instruction `storage_diff`, which the translator adds as the last instruction to every Michelson contract.

This `storage_diff` is specialized in the translation phase, as it is included in the intermediate representation of our Michelson compiler right before the translation process begins. Thus, our Michelson partial evaluator is the one in charge of specializing this instruction using this simple rule:

$$\texttt{storage\_diff}(\texttt{t}, \texttt{storage}, \texttt{storage}') = \begin{cases} (\texttt{storage\_diff}(\texttt{ta}, \texttt{car}(\texttt{storage}), \texttt{car}(\texttt{storage}')), \\ \ \texttt{storage\_diff}(\texttt{tb}, \texttt{cdr}(\texttt{storage}), \texttt{cdr}(\texttt{storage}'))) & \text{if } \texttt{t} = (\texttt{pair ta tb}) \\ \texttt{storage\_diff}_{\texttt{t}}[\texttt{storage}, \texttt{storage}'] & \text{otherwise} \end{cases}$$

Where ',' represents the conjunction. This way, Equation 4.6 is preserved, as, if the storage type is an n-tuple, we generate n specialized `storage_diff` calls in the output CHC IR code, whose storage cost is accumulated by adding them together.

In addition to including them in the code, we must also define the cost semantics for these cost markers in our cost models. In Listing 4.14, we can see assertions for these new cost markers. Storage difference expressions for `bool` and `string` types are trivial, as `bool` values are encoded in a single byte and `strings` as a list of bytes. On the other hand, the storage difference expression for `int` (arbitrary precision integers) is more complex, due to its particular encoding.

Storage expressions for cost markers of most types, e.g., `bool`, `string` or `int`, are simple enough to not need a previous treatment. On the other hand, Michelson also provides *structural types*, e.g., `list`, `set`, `map`, which encode data structures of a given type. To handle storage markers for this kind of types, we must add an

```
:- trust pred '$storage_diff_list'(A,B,C)
   + ( cost(lb,michelson_storage,(length(B)-length(A))*int(C)),
       cost(ub,michelson_storage,(length(B)-length(A))*int(C)) ).
```

Listing 4.15: Assertion for `list michelson_storage` cost marker.

```
:- trust pred '$storage_diff_top'(A,B)
   + cost(ub,michelson_storage,inf).
```

Listing 4.16: $\top$ `michelson_storage` cost marker.

additional input argument to the cost markers: the size of the terms in the structure. Listing 4.15 shows the storage expression for a generic list, where `C` is the size of each element in the list. Given `sizeof` as a partial function which returns the size (in bytes) of the elements of a given type iff every element of this type has the same size:

$$michelson\_storage((\texttt{list t}), \Delta_i(parameter, storage)) = \Delta_i(parameter, storage) * \texttt{sizeof(t)} \tag{4.7}$$

E.g., for a list of `bool` ($\texttt{sizeof(bool)} = 1$), we would have a storage cost expression:

$$michelson\_storage((\texttt{list bool}), \Delta_i(parameter, storage)) = \Delta_i(parameter, storage)$$

Now, with `sizeof` a partial function, we may find cases in which Equation 4.7 is not defined. Thus, we need to handle this case:

$$michelson\_storage((\texttt{list t}), \Delta_i(parameter, storage)) = \begin{cases} \Delta_i(parameter, storage) * \texttt{sizeof(t)} & \text{if } \texttt{sizeof(t)} \text{ is defined} \\ \top & \text{otherwise} \end{cases} \tag{4.8}$$

When $michelson\_storage$ is input a type whose size difference expression cannot be known statically, i.e., a structural type containing elements of a type for which `sizeof` is not defined, e.g., `(list int)`, the output cost marker will be that defined in Listing 4.16. This is the "top" ($\top$) cost marker, which has a storage consumption in the range $[0, \infty)$.

Please note that, using a more advanced size analysis, like the *sized types* analysis of [55], we could improve the precision of michelson storage resource analysis. Modifying Equation 4.8, we would obtain:

$$michelson\_storage((\texttt{list t}), L, l, M, m, L', l', M', m') \in [(l' * \texttt{sizeof(t}, m')) - (L * \texttt{sizeof(t}, M)), (L' * \texttt{sizeof(t}, M')) - (l * \texttt{sizeof(t}, m))]$$

Where `sizeof` has been overloaded to represent the size of a concrete term given its type and value, L/L', resp. l/l', is the maximum, resp. minimum, possible length for the input/output list; and M/M', resp. m/m', is the term with the maximum, resp. minimum, size in the input/output storage. Note that, for the case in which $\texttt{sizeof(t)}$ is defined, the expression is equivalent to that in Equation 4.8—as we would have that $\texttt{sizeof(t}, m) = \texttt{sizeof(t}, m') = \texttt{sizeof(t}, M) = \texttt{sizeof(t}, M') = \texttt{sizeof(t)}$.

4.3.2. Detaching Semantics from Cost Semantics

After writing the Delphi cost model, we found out that information regarding the instruction semantics was duplicated in both cost models. In order to tackle this problem, we decided to split the semantics and cost semantics definitions into two different sets of assertions. Exploiting the Ciao packages system, we created a new package containing assertions that declare instructions semantics which is loaded by every cost model package.

By implementing this change to our cost models, we were able to reduce the size of our cost models by 32 % (1490 LOC). As properties expressing semantics and cost semantics are disjoint and semantics for Ciao assertions are assumed to be $\top$ for missing properties, we have that the greatest lower bound ($\sqcap$) of two different assertions (one defining the semantics of an instruction and the other, its cost semantics) for a given instruction is:

$$\left\{ \begin{array}{ll} semantics & : s \\ cost & : \top \end{array} \right\} \sqcap \left\{ \begin{array}{ll} semantics & : \top \\ cost & : c \end{array} \right\} = \left\{ \begin{array}{ll} semantics & : s \sqcap \top \\ cost & : \top \sqcap c \end{array} \right\} = \left\{ \begin{array}{ll} semantics & : s \\ cost & : c \end{array} \right\}$$

So, the system can handle this separation of semantics and cost semantics, saving a great amount of time and space, the main motivation of Parametric Cost Analysis.

4.4. Improvements to Michelson Partial Evaluation

As we previously stated in Section 3.1, our interpreter is capable of unfolding control flow instructions if the value of the condition is known statically. Thus, analysis precision could be far improved by extending the partial evaluation capabilities of our analysis.

In Listing 4.10, we can see how the NIL instruction is executed statically, as its output will always be an empty list. By taking a similar approach, we can evaluate the outputs of some instructions if their input arguments are sufficiently instantiated.

Control flow instruction unfolding is achieved by providing some *metainformation*, describing the nature of the input arguments to a higher-order instruction, i.e., if they are code, a branching condition or another term. In a similar fashion, we can write *partial evaluation rules*, which express the *conditions* under which an instruction can be partially evaluated by our interpreter. We have included two types of partial evaluation conditions: *instantiation conditions* and *relation conditions*.

Instantiation conditions are lists of conditions input arguments must fulfill for an instruction to be partially evaluated. We will show each condition we have considered by taking a look at the rules for the ADD instruction:

```
pe_mode(add, [pe_const(0),term,-]).
pe_mode(add, [term,pe_const(0),-]).
pe_mode(add, [+,+,-]).
```

As the ADD instruction has two input arguments and one output argument, the lists encoding partial evaluation conditions have three elements, one for each argument. Output arguments are marked as "-", meaning that no test has to be performed on them, as they will always be free variables. An equivalent condition to "-" is "term",

which is used for input arguments which can be any term (ground or free) on input. Finally, the "+" and "`pe_const(X)`" conditions are used to state that a value must be ground on input or that it must be instantiated to a certain value `X`, respectively. As the `ADD` rules encode, this instruction can be partially evaluated if any of the following conditions hold:

- The first input is 0
- The second input is 0
- Both inputs are ground

Some instructions need additional conditions, such in the case of `CONCAT` instruction, which concatenates a list of strings. Rules for this instruction include the `concat_string` condition, stating that the input must be a list of ground terms with a (possibly) free tail, e.g., `["Ciao","Michelson"]` or `["Ciao","PP"|_]`.

Regarding relation conditions, these are used to express a relation that must hold between two or more input arguments. In the case of the `SUB` instruction, it includes the following rules:

```
pe_mode(sub, [term,pe_const(0),-]).
pe_mode(sub, eq(1,2)).
pe_mode(sub, [+,+,-]).
```

The second rule introduces a new condition: `eq(1,2)`, which states that arguments in positions 1 and 2 must be *strictly identical*, i.e., ground and with the same value, the same variable or a compound term with the same functor and identical arguments. Thus, this instruction can be partially evaluated if any of the following conditions hold:

- The second input is 0
- Both inputs are strictly identical
- Both inputs are ground

Using these partial evaluation rules, an artificial Michelson contract like:

```
1  parameter int ;
2  storage int ;
3  code { UNPAIR ;
4         PUSH int 0 ;
5         MUL ;
6         ADD ;
7         DUP ;
8         SUB ;
9         PUSH int 10 ;
10        ADD ;
11        NIL operation ;
12        PAIR }
```

Can be translated to a CHC IR representation in which the output is known at compile time:

```
default(A,B,[],10) :-
    '$unpair', '$const'(0),
    mul(0,A,0),
    add(0,B,B),
    '$dup',
    sub(B,B,0),
    '$const'(10),
    add(10,0,10),
    nil([]),
    '$cons_pair'.
```

In addition to these partial evaluation rules, we include *compare* rules. These rules are used to improve the precision and readability of the translation by introducing metainformation in the translation stack when two terms are compared. Thus, if a comparison instruction followed by a branching instruction is found in a contract, such as in this example,[3] which stores the absolute value of the parameter in the storage:

```
parameter int ;
storage int ;
code { CAR ; DUP ; LT ; IF { NEG } {} ; NIL operation ; PAIR }
```

We can make the branching instruction clearer—also helping the analysis—by including a comparison literal `A < B` in the body of the generated predicate encoding the branch instruction:

```
default(A, B, [], C) :-
    '$car', '$dup',
    lt(A, D),
    '$if',
    if__0(D, A, A, C),
    nil([]),
    '$cons_pair'.

if__0(true, A, B, C) :- A < 0,
    neg(B, C).
if__0(false, A, B, B) :- A >= 0.
```

4.5. Entrypoints Support

Michelson contracts can implement different functionalities thanks to the *entrypoints* mechanism. A contract with entrypoints is a Michelson contract which receives a parameter of type `(or ta tb)`, `ta` and `tb` being possibly nested disjunctive types, i.e., more `(or ta' tb')` types. This input parameter type must be annotated in the source code using *entrypoints*—also called *field*—annotations, with shape "`%Annot`".

Using this contract polymorphism mechanism, we can extend the capabilities of Michelson smart contracts, e.g., the smart contract in Listing 1.1 can be extended

[3] LT is an implicit comparison with 0.

so that it provides entrypoints to different list operations to modify the stored list: it could reverse the stored list, append an element to it, clear it or even apply an input function to each of its elements, storing the result in the storage list. These operations can be implemented as follows:

```
parameter
  (or (or (or (lambda %map mutez mutez) (mutez %cons))
  (unit %reverse)) (unit %default)) ;
storage (list mutez) ;
code { UNPAIR ;
       IF_LEFT
         { IF_LEFT
             { IF_LEFT
                 { SWAP ; MAP { DIP { DUP } ; EXEC } ;
                   DIP { DROP } }
                 { CONS } }
             { DROP ; NIL mutez ; SWAP ; ITER { CONS } } }
         { DROP 2 ; NIL mutez } ;
       NIL operation ;
       PAIR }
```

As we can see, this smart contract provides four different entrypoints: *1)* `map`, receiving an input lambda function to apply to the elements of the list; *2)* `cons`, receiving the element to append to the list; *3)* `reverse`, which receives a `unit` (empty) value and reverses the list; and *4)* `default`, which receives a `unit` (empty) value and clears the contents in the list.

As Michelson entrypoints are syntactic sugar over disjunctive types restricting that entrypoint annotations cannot be duplicated in a parameter type definition, we can use the partial evaluation capabilities of our interpreter to residualize the contract code for each of the possible parameter types. Thus, in the previous example, we should take into account all the possible parameter types annotated by entrypoints[4] and generate a different predicate for each one of these. In Listing A.1, we can find the CHC IR representation of the contract in Listing 4.5. As seen in Listing 4.17, our translation tool not only generates a different predicate for each entrypoint, but also, a set of assertions describing the types of the input terms to each entrypoint.

4.6. Defining a Michelson Assertion Language

When dealing with verification of a programming language, it is important to define the way in which the tool and the programmer will communicate with each other. `Ciao`'s approach is defining an *assertion language* so the user can input information about the predicates to `CiaoPP`, which will be able to communicate the inferred relevant information back to the user using the same mechanism. In Section 4.6.1, we will briefly introduce `Ciao`'s assertion language and in Section 4.6.2, we will show a first prototype of an assertion language for Michelson.

[4] All the possible combinations are: `left (left (left (lambda mutez mutez)))`, `left (left (right mutez))`, `left (right unit)` and `right unit`.

```
:- entry cons(A, B, C, D)
   : ( mutez(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) ).
cons(A, B, [], [A|B]) :- ...

:- entry map(A, B, C, D)
   : ( lambda(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) ).
map(A, B, [], C) :- ...

:- entry reverse(A, B, C, D)
   : ( unit(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) ).
reverse('()', A, [], B) :- ...

:- entry default(A, B, C, D)
   : ( unit(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) ).
default('()', A, [], []) :- ...
```

Listing 4.17: Head of the predicates generated for each entrypoints in Listing 4.5.

4.6.1. Introduction to Ciao's Assertion Language

Ciao assertions [56, 13, 57] can be used to express different properties of predicates in the source code, be it functional properties, e.g., types, modes, aliasing; or non-functional, e.g., non-failure, determinacy, cost. Assertions in Ciao are a bidirectional communication channel between programmer and CiaoPP, so the former can use them to write specifications, describing unknown code or expressing properties that a certain part of the code must fulfill; whereas the analysis tool will use them to report analysis and verification results back to the user.

We will focus on pred assertions, used to describe a set of *preconditions* and *postconditions* on a predicate in the source code. These assertions are of the form:

```
:- [ Status ] pred Head [:  Pre ] [=> Post ] [+ Comp ].
```

where Head is a descriptor of the predicate to which the assertion applies and Pre and Post are conjunction of *property literals*. These properties are predicates which can also be used as run-time checks and can be abstracted and inferred by some domain in CiaoPP. Pre, resp. Post, express properties which must hold when Head is called, resp. when Pre holds and the call succeeds. If Pre or Post are empty (true), they can be omitted. Regarding the Comp field, it can be used to express properties of the computation, e.g., cost, termination, determinism, etc., and they apply to calls of the predicate that meet Pre. Finally, Status is a keyword expressing the meaning of the assertion, which can be any of:

check (default, can be omitted) the assertion expresses properties which must hold at run-time, so the static analyzer should be able to prove them or generate run-time checks for them;

checked the analyzer proved that the property holds;

false the analyzer proved that the property does not hold in some execution;

trust used to specify properties of the code.

Thus, when we use check assertions in the code, CiaoPP will use check, checked or false assertions to express that properties must be further checked, hold or do not hold. E.g., for the following predicate append/3, we could write:

```
:- pred append(Xs, Ys, Zs) : ( list(Xs), list(Ys) ) => list(Zs).
:- pred append(Xs, Ys, Zs) : list(Zs) => ( list(Xs), list(Ys) ).

append([],     Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).
```

This check assertions express that, if we call append/3 using two lists in positions 1 and 2 of the head, the third argument must be a list on output; and the other way around. As these assertions hold and can be checked statically by CiaoPP, the output of the verification tool when receiving this program as an input is:

```
:- checked append(Xs, Ys, Zs) : ( list(Xs), list(Ys) ) => list(Zs).
:- checked append(Xs, Ys, Zs) : list(Zs) => ( list(Xs), list(Ys) ).

append([],     Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).
```

Regarding trust assertions, they can be used to specify properties for an unknown predicate or, as seen in Section 3.2, to define a cost model.

4.6.2. The CiaoMichelson Assertion Language

Although we have found a powerful tool in Ciao assertions to express predicates properties, we currently do not have a way to include such assertions in Michelson contracts. A possible solution to this problem would be to manually add assertions to the CHC IR generated by our translation tool, but this may be a slow process when contracts are being updated regularly.

Our first approach to expressing computational properties in Michelson code involved using special Michelson comments to write inline Ciao assertions in Michelson, but this method felt alien in a Michelson contract and was thus far from ideal. To fix this issue, we have defined and implemented an extension of the Michelson language, CiaoMichelson, which is Micheline[5] compliant and uses native structures to express program properties.

As we know, Michelson contracts count with three toplevel fields: parameter, storage and code. Our approach to defining a Michelson assertion language is including a new type of field, assert, of the form:

```
assert [% entrypoint] { [property,]* }
```

Where entrypoint is the name of an entrypoint in the code (default by default) and property is a functional or non-functional property expressed in a Michelson-native

[5]Micheline is the syntax used in Michelson.

language. These assert assertions correspond to check assertions in the Ciao system, so they should be verified by CiaoPP.

Currently, we have only defined a Michelson property: output_size, which can be used to express the sizes of the output storage in terms of the input parameter and storage. This property has the following shape:

```
output_size { [instruction,]+ }
```

Where instruction is an instruction belonging to a set of Michelson instructions needed to perform basic operations such as dropping elements from the stack, pushing new values, performing arithmetic operations, etc., and four additional operations used to cover some of the arithmetic operations which can be used in Ciao assertions but not in Michelson code:

DIV Division with no zero checks;

POW Exponentiation;

LN Natural logarithm;

LOG Logarithm receiving the base as an additional argument.

For a given entrypoint ep, output_size code holds when:

$$\forall \texttt{Parameter} \in \texttt{parameter}, \texttt{Storage} \in \texttt{storage}, \texttt{output\_size}(\texttt{ep}, \texttt{Parameter}, \texttt{Storage}, \texttt{code})$$
$$\iff \texttt{sizeof}(\texttt{ep}(\texttt{Parameter}, \texttt{Storage})) = \texttt{code}(\texttt{Pair sizeof}(\texttt{Parameter})\ \texttt{sizeof}(\texttt{Storage}))$$
$$\textit{where } \texttt{code} : \texttt{sizof}(\texttt{pair parameter storage}) \rightarrow \texttt{sizeof}(\texttt{storage}),$$
$$\texttt{sizeof}(\texttt{t}) = \begin{cases} (\texttt{pair sizeof}(\texttt{ta})\ \texttt{sizeof}(\texttt{tb})) & \text{if } \texttt{t} = (\texttt{pair ta tb}) \\ \texttt{int} & \text{otherwise} \end{cases}$$
$$\textit{and } \texttt{ep}(\texttt{Parameter}, \texttt{Storage}) \text{ is the resulting storage when executing entrypoint ep with arguments Parameter and Storage}$$

Even though sizeof(Parameter) and sizeof(Storage) are symbolic representations of the sizes of the input arguments to a contract, we could use our interpreter to generate code for the body of an output_size property. However, this is not what we want, as this code is supposed to be translated to a valid Ciao arithmetic expression to be used in the body of an assertion. Thus, in order to generate a Ciao property from an output_size property, we have defined a set of rules which can be used with our Michelson interpreter to automatically generate the desired properties.

To show this functionality in action, we are going to add correct Michelson assertions to the contract in Listing 4.5. For the map entrypoint, we have that the size of the list is not modified, so our output_size property should express that; cons increases the length of the list in one unit, reverse does not change the length of the list and default inserts an empty list (of size 0) in the storage:

```
assert %map { output_size { CDR } };
assert %cons { output_size { CDR ; PUSH nat 1 ; ADD } };
assert %reverse { output_size { CDR } };
assert %default { output_size { DROP ; PUSH nat 0 } }
```

The output Ciao assertions for these Michelson assertions are:

```
:- check pred map(A, B, C, D)
   : ( lambda(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   => ( lambda(A), list(mutez, B), list(operation, C),
        list(mutez, D), ground(A), ground(B), ground(C),
        ground(D), size(D, length(B)) ).

:- check pred cons(A, B, C, D)
   : ( mutez(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   => ( mutez(A), list(mutez, B), list(operation, C),
        list(mutez, D), ground(A), ground(B), ground(C),
        ground(D), size(D, 1 + length(B)) ).

:- check pred reverse(A, B, C, D)
   : ( unit(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   => ( unit(A), list(mutez, B), list(operation, C),
        list(mutez, D),
        ground(A), ground(B), ground(C), ground(D),
        size(D, length(B)) ).

:- check pred default(A, B, C, D)
   : ( unit(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   =>( unit(A), list(mutez, B), list(operation, C),
       list(mutez, D),
       ground(A), ground(B), ground(C), ground(D),
       size(D, 0) ).
```

As we can see, these assertions perfectly reflect the meaning of the assertions written in Michelson, adding extra types and instantiation information to be used by the analyses. If we try to verify these assertions using CiaoPP, the tool will output a set of assertions with checked status, meaning that the assertions hold:

```
:- checked pred map(A, B, C, D)
   : ( lambda(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   => ( lambda(A), list(mutez, B), list(operation, C),
        list(mutez, D), ground(A), ground(B), ground(C),
        ground(D), size(D, length(B)) ).

:- checked pred cons(A, B, C, D)
   : ( mutez(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   => ( mutez(A), list(mutez, B), list(operation, C),
        list(mutez, D), ground(A), ground(B), ground(C),
        ground(D), size(D, 1 + length(B)) ).
```

```
:- checked pred reverse(A, B, C, D)
   : ( unit(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   => ( unit(A), list(mutez, B), list(operation, C),
        list(mutez, D),
        ground(A), ground(B), ground(C), ground(D),
        size(D, length(B)) ).

:- checked pred default(A, B, C, D)
   : ( unit(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) )
   =>( unit(A), list(mutez, B), list(operation, C),
       list(mutez, D),
       ground(A), ground(B), ground(C), ground(D),
       size(D, 0) ).
```

Chapter 5

CiaoPP Improvements

In this section, we will go through the improvements that were implemented in the CiaoPP analysis tool to improve the precision of the analysis. These were motivated by use cases that were found during the development of the ciao_tezos tool. First, in Section 5.1, we will introduce a new approach to compound resources that was included in CiaoPP to deal with Tezos gas. Section 5.2 covers a solution we used to deal with failing clauses when performing size analysis of smart contracts. Finally, in Section 5.3, we will discuss some of the improvements to CiaoPP's built-in linear equation solver.

5.1. Improvements to Compound Resources

A *compound resource* is a resource which can be expressed in terms of other *atomic* (or compound) resources, as it is the case for Tezos gas (see Section 3.2). Although CiaoPP has supported compound resources for a long time, this support was somewhat limited, and had to be extended/improved to handle some of Tezos gas' particularities. In Section 5.1.1, we cover our new approach to defining compound resources as a mathematical expression and in Section 5.1.2, we introduce the new, more efficient, way in which cost expressions for compound resources are obtained in CiaoPP.

5.1.1. Defining Compound Resources as Mathematical Expressions

Early implementations of compound resources in CiaoPP only allowed defining a compound resource as the dot product of a *resources vector* (**res**) and a constants (also referred to as *platform*) vector (**k**):

$$compound\_resource(\mathbf{res}, \mathbf{k}) = \mathbf{res} \cdot \mathbf{k} = \begin{bmatrix} res_1 & \cdots & res_n \end{bmatrix} \begin{bmatrix} k_1 \\ \vdots \\ k_n \end{bmatrix} = \sum_{i=1}^{n} res_i * k_i \tag{5.1}$$

This was powerful enough in some cases, but would fall short in the general case. This is why we decided to allow the definition of compound resources as a mathematical expression in terms of other resources:

$$compound\_resource(\mathbf{res}) = f(\mathbf{res}) \tag{5.2}$$

```
:- compound_resource(michelson_gas,
                     [atomic_steps, michelson_allocations,
                        michelson_reads, michelson_writes,
                        michelson_bytes_read,
                        michelson_bytes_written]).

:- platform_constants(delphi, michelson_gas,
                      [1, 2000, 1000, 100000, 160000, 10000,
                         15000]).
```

Listing 5.1: Definition of Tezos gas as a compound resource previous to language extension.

This way, compound resources can be expressed in a more powerful, convenient, and readable format. In Listing 5.1, we define Tezos gas in the Delphi protocol as a compound resource using legacy syntax, whereas in Listing 4.1, we can see how using the new syntax results in a clearer assertion.

In addition to clarity, this new syntax makes use of a rich arithmetic language which was also extended to better express the cost semantics of the Tezos platform. Some of the arithmetic expressions included in this language are the summation (`sum`), product of a sequence (`prod`), exponentiation (`**`), trigonometric expressions (`sin`, `cos`, `tan`) and some fundamental mathematical constants (`e`).

An important aspect to take into account is that, although we allow the definition of compound resources in terms of other compound resources, we do not allow the recursive definition of these elements, as this would require solving an additional recurrence relation. To enforce this constraint, an ad hoc circularity analysis is performed on the definition of the compound resources in order to detect recursive compound resources definitions.

5.1.2. A "Global" Approach to Compound Resources Calculation

By compound resources definitions in Equations 5.1 and 5.2, we can infer that the cost inferred for a compound resource must be equal to the application of the definition of the compound resource to the inferred costs for its component resources:

$$compound\_resource(\mathbf{res}) = f(\mathbf{res}) \iff cost(compound\_resource) = f(cost(\mathbf{res}))$$

where *cost* is a function that maps (vectors of) resources to (vectors containing) their costs. This opens two possibilities when calculating compound resources: *1)* a *local* approach, already previously supported in `CiaoPP`, which accumulates the cost of each literal in terms of the compound resource; or *2)* a *global* approach, new for `CiaoPP`, which calculates only the cost of atomic resources from which the cost of compound resources can be obtained.

The local approach is more costly, not only due to the additional operations that must be performed in order to accumulate the cost of each literal in the code and set up the recurrence relations which give the cost for each compound resource, but also due to the increment in the number of recurrence relations to be solved in the solving stage.

On the other hand, the global approach is cheaper, as the number of operations and recurrence relations to solve is lower, but there might be some cases in which this method is less precise. E.g., if we run into a situation in which two atomic resources (res_i, res_j) happen to have exactly the same cost and both cancel out:

$$\left.\begin{array}{r} compound\_resource(\mathbf{res}) = f(\mathbf{res}) = g(\mathbf{res} \setminus \{res_i, res_j\}) + res_i - res_j \\ cost(res_i) = cost(res_j) \\ res_i, res_j \in \mathbf{res} \end{array}\right\} \implies$$
$$compound\_resource(\mathbf{res}) = g(\mathbf{res} \setminus \{res_i, res_j\})$$

There could be a program in which the recurrence relation solver is unable to find an expression for res_i and res_j. In this case, trying to obtain the cost of the compound resource using the global approach would be infeasible; only the local approach, in which the inferred cost of both resources may cancel out as the cost is accumulated, would be useful in this particular scenario.

5.2. Dealing with Failing Clauses during Size Analysis

The CiaoPP size analysis accumulates size information for the different clauses of a predicate by obtaining the least upper bound ($\bigsqcup$) of the size intervals inferred for each clause, i.e., being X a variable, P a predicate in the code, $P_1, \ldots, P_n$ the n clauses of predicate P ($n > 0$), $size(X, P)$ the inferred size of variable X for predicate/clause P and $ub(Int)$/$lb(Int)$ the upper/lower bound of an interval Int:

$$size(X,P) = \bigsqcup_{i=1}^{n} size(X,P_i) = \begin{cases} \bot & \text{if } \exists i, \bot \in P_i \\ [\min_i \{lb(size(X,P_i))\}, \max_i \{ub(size(X,P_i))\}] & \text{otherwise} \end{cases} \tag{5.3}$$

However, this definition leads to a big loss of precision when we run into failure. For a clause P_i, of predicate P, if P_i surely fails,[1] we get that:

$$\exists i \text{ s.t. } P_i \text{ fails} \implies \forall X, size(X,P_i) = \bot \implies \forall X, size(X,P) = \bot \text{ by 5.3}$$

If we go back to the definition of `Post` properties in `Ciao`, in Section 4.6.1, we can see that these properties must only hold *when the predicate succeeds*. Thus, from the definition of `Post` properties themselves, we can infer that the definition given by Equation 5.3 is incorrect.

In order to fix this, we must redefine the $\bigsqcup$ operation:

$$size(X,P) = \bigsqcup_{i=1}^{n} size(X,P_i) =$$
$$\left[\min_i \{lb(size(X,P_i)) \mid P_i \text{ may not fail}\}, \max_i \{ub(size(X,P_i)) \mid P_i \text{ may not fail}\}\right] \tag{5.4}$$

Using this new definition, if a clause P_i of predicate P surely fails, we get:

$$\exists i \text{ s.t. } P_i \text{ fails} \implies size(X,P) = \left[\min_j \{lb(size(X,P_j)) \mid i \neq j\}, \max_j \{ub(size(X,P_i)) \mid i \neq j\}\right] \text{ by 5.4}$$

[1] In `ciao_tezos`, this is a common pattern, as we are encoding Michelson failure as a logic failure.

```
:- true pred p(X, Y)
   : ( int(X), var(Y) )
   => ( int(X), int(Y),
        size(lb, int, Y, int(X)) )
   + cost(lb, steps, 0).

:- true pred p(X, Y)
   : ( int(X), var(Y) )
   => ( int(X), int(Y),
        size(ub, int, Y, int(X)) )
   + cost(ub, steps, 1).

p(X, X).

q(X, _) :- X < 0, !, fail.
q(X, Y) :-
    Y is X + 1.
```

Listing 5.2: Analysis of a predicate with a failing clause.

Using this new definition of the $\bigsqcup$ operation, we can analyze simple programs like that in Listing 5.2.

5.3. Extending the Built-in Recurrence Relations Solver Capabilities

In order to obtain cost expressions showing resource consumption of programs, the analyzer has to solve the recurrence relations obtained in previous steps, using an external solver or the built-in recurrence relations solver. When trying to analyze some programs, the analysis may set up the equations correctly, but fail to solve them. In this case, the recurrence relations solver must be extended to support the new kind of equations. In our case, we found out that recurrence relations of the following shape could not be solved by our built-in solver:

$$f\left(x,Y,Z\right)=\begin{cases}g(Y,Z) & \text{if } x=0\\ uf(x-1,Y+\Delta_Y,K\circ Z+\Delta_Z)+h(x,Y,Z) & \text{if } x>0\end{cases},(\forall k\in K.k\neq 1) \tag{5.5}$$

where $\circ$ is the element-wise vector multiplication.

In order to support this kind of equations in our solver, we must first find a general solution for it and encode it. We assume that the following expression is a solution to the recurrence relation—being $Z^*=\Delta_Z\oslash(J-K)$, $X^{\circ n}$, the element-wise vector exponentiation; $\oslash$, the element-wise vector division; and J, the vector of ones:

$$\begin{aligned}f(x,Y,Z)=&u^x g\left(Y+(x\Delta_Y),K^{\circ x}\circ(Z-Z^*)+Z^*\right)+\\ &\sum_{i=0}^{x-1}u^i h\left(x-i,Y+(i\Delta_Y),K^{\circ i}\circ(Z-Z^*)+Z^*\right)\end{aligned} \tag{5.6}$$

We are going to prove that this is a correct solution by induction on x. First, we will

test the solution for $x = 0$ and $x = 1$:

$$\begin{aligned} f(0,Y,Z) =& u^0 g\left(Y + (0\Delta_Y), K^{\circ 0} \circ (Z - Z^*) + Z^*\right) + \\ & \sum_{i=0}^{0-1} u^i h\left(0-i, Y + (i\Delta_Y), K^{\circ i} \circ (Z - Z^*) + Z^*\right) \\ =& g(Y,Z) \end{aligned} \tag{5.7}$$

$$\begin{aligned} f(1,Y,Z) =& u^1 g\left(Y + (1\Delta_Y), K^{\circ 1} \circ (Z - Z^*) + Z^*\right) + \\ & \sum_{i=0}^{1-1} u^i h\left(1-i, Y + (i\Delta_Y), K^{\circ i} \circ (Z - Z^*) + Z^*\right) \\ =& ug(Y + \Delta_Y, K \circ Z + \Delta_Z) + h(1,Y,Z) \end{aligned} \tag{5.8}$$

Now, we assume that the solution is satisfied for $x = n - 1$:

$$\begin{aligned} f(n-1,Y,Z) =& u^{n-1} g\left(Y + ((n-1)\Delta_Y), K^{\circ(n-1)} \circ (Z - Z^*) + Z^*\right) + \\ & \sum_{i=0}^{n-2} u^i h\left(n-1-i, Y + (i\Delta_Y), K^{\circ i} \circ (Z - Z^*) + Z^*\right) \end{aligned} \tag{5.9}$$

We can test that, if the solution holds for $x = n - 1$, the solution must hold for $x = n$:

$$\begin{aligned} f(n,Y,Z) &= uf(n-1, Y + \Delta_Y, K \circ Z + \Delta_Z) + h(n,Y,Z) \\ &= uu^{n-1} g\left((Y + \Delta_Y) + ((n-1)\Delta_Y), K^{\circ(n-1)} \circ ((K \circ Z + \Delta_Z) - Z^*) + Z^*\right) \\ &+ u \sum_{i=0}^{n-2} u^i h\left(n-1-i, (Y + \Delta_Y) + (i\Delta_Y), K^{\circ i} \circ ((K \circ Z + \Delta_Z) - Z^*) + Z^*\right) \\ &+ h(n,Y,Z) \\ &= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ Z + K^{\circ(n-1)} \circ (\Delta_Z - Z^*) + Z^*\right) \\ &+ \sum_{i=0}^{n-2} u^{i+1} h\left(n-1-i, Y + ((i+1)\Delta_Y), K^{\circ(i+1)} \circ Z + K^{\circ i} \circ (\Delta_Z - Z^*) + Z^*\right) \\ &+ h(n,Y,Z) \\ &= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ Z + K^{\circ(n-1)} \circ (\Delta_Z - \Delta_Z \circ (J - K)) + Z^*\right) \\ &+ \sum_{i=0}^{n-2} u^{i+1} h\left(n-1-i, Y + ((i+1)\Delta_Y), K^{\circ(i+1)} \circ Z + K^{\circ i} \circ (\Delta_Z - \Delta_Z \circ (J - K)) + Z^*\right) \\ &+ h(n,Y,Z) \\ &= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ Z + K^{\circ(n-1)} \circ (-K \circ \Delta_Z \oslash (J - K)) + Z^*\right) \\ &+ \sum_{i=0}^{n-2} u^{i+1} h\left(n-1-i, Y + ((i+1)\Delta_Y), K^{\circ(i+1)} \circ Z + K^{\circ i} \circ (-K \circ \Delta_Z \circ (J - K)) + Z^*\right) \\ &+ h(n,Y,Z) \\ &= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ Z - K^{\circ n} \circ \Delta_Z \oslash (J - K) + Z^*\right) \\ &+ \sum_{i=0}^{n-2} u^{i+1} h\left(n-1-i, Y + ((i+1)\Delta_Y), K^{\circ(i+1)} \circ Z - K^{\circ(i+1)} \circ \Delta_Z \circ (J - K) + Z^*\right) \end{aligned}$$

$$
\begin{aligned}
&+ h(n, Y, Z) \\
&= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ Z - K^{\circ n} \circ Z^* + Z^*\right) \\
&+ \sum_{i=0}^{n-2} u^{i+1} h\left(n-1-i, Y + ((i+1)\Delta_Y), K^{\circ(i+1)} \circ Z - K^{\circ(i+1)} \circ Z^* + Z^*\right) \\
&+ h(n, Y, Z) \\
&= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ (Z - Z^*) + Z^*\right) \\
&+ \sum_{i=0}^{n-2} u^{i+1} h\left(n-1-i, Y + ((i+1)\Delta_Y), K^{\circ(i+1)} \circ (Z - Z^*) + Z^*\right) \\
&+ h(n, Y, Z) \\
&= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ (Z - Z^*) + Z^*\right) \\
&+ \sum_{i=1}^{n-1} u^{i} h\left(n-i, Y + (i\Delta_Y), K^{\circ i} \circ (Z - Z^*) + Z^*\right) \\
&+ u^0 h\left(n-0, Y + (0\Delta_Y), K^{\circ 0} \circ (Z - Z^*) + Z^*\right) \\
&= u^n g\left(Y + (n\Delta_Y), K^{\circ n} \circ (Z - Z^*) + Z^*\right) \\
&+ \sum_{i=0}^{n-1} u^{i} h\left(n-i, Y + (i\Delta_Y), K^{\circ i} \circ (Z - Z^*) + Z^*\right) \qquad (5.10)
\end{aligned}
$$

By Equations (5.7) and (5.8), we can see that this solution is satisfied when $n = 0$ or $n = 1$ and by Equations (5.9) and (5.10), we can prove that, when n satisfies the solution, $n + 1$ also does. Thus, we can state that Equation 5.6 is a solution for the recurrence relation in Equation 5.5.

In Listing 5.3, we include an example of a program which can be analyzed using the recurrence relation in Equation 5.6. It receives two sets of numbers, Xs and Ys; and a number, R, as inputs; and outputs the following set:[2]

$$Zs = Ys \cup \{2^i * R * x_i \mid 0 \leq i < |X|\}$$

We assume that the cost in computational steps of inserting an element in a set is linear with respect to the length of the set and that the cost of multiplying two numbers is logarithmic with respect to the value of the second number. With this information, we can set up a system of equations with the shape of that in Equation 5.5:

[2]We assume that sets are encoded as sorted lists.

```
:- entry union_prod(Xs, Ys, R, Zs)
   : ( list(num, Xs), list(num, Ys), var(Zs), num(R) ).

union_prod([], Ys, _, Ys).
union_prod([X|Xs], Ys0, R, Zs) :-
    prod(X, R, X1),
    prod(R, 2, R1),
    insert_(Ys0, X1, Ys1),
    union_prod(Xs, Ys1, R1, Zs).

:- impl_defined(insert_/3).

:- trust pred insert_(A,B,C)
   : ( list(num,A), num(B), var(C) )
   => ( list(num,A), num(B), list(num,C),
        size(lb,C,length(A)), size(ub,C,length(A)+1) )
   + ( not_fails, is_det, mut_exclusive, covered,
       cost(lb,steps,1),
       cost(ub,steps,length(A)+1) ).

:- impl_defined(prod/3).

:- trust pred prod(A,B,C)
   : ( num(A), num(B), var(C) )
   => ( num(A), num(B), num(C),
        size(lb,C,int(A)*int(B)),
        size(ub,C,int(A)*int(B)) )
   + ( not_fails, is_det, mut_exclusive, covered,
       cost(lb,steps,1),
       cost(ub,steps,log2(int(B))) ).
```

Listing 5.3: Program analyzable using the recurrence relation in Equation 5.6.

$$f(x,y,r) = \begin{Bmatrix} 1 & \text{if } x = 0 \\ f(x-1, y+1, 2*r) + y + \log_2 r + 3 & \text{if } x > 0 \end{Bmatrix} \implies$$

$$\begin{cases} Y = [y] \\ \Delta_Y = [1] \\ Z = [r] \\ \Delta_Z = [0] \\ K = [2] \\ u = 1 \\ g(Y,Z) = 1 \\ h(x,Y,Z) = y + \log_2 r + 3 \end{cases}$$

Using the formula in Equation 5.6, we find that the solution for this system is:

$$f(x,y,r) = x^2 + xy + x\log_2 r + 2x + 1$$

```
:- true pred union_prod(Xs,Ys,R,Zs)
   : ( list(num,Xs), list(num,Ys), num(R), var(Zs) )
   => ( list(num,Xs), list(num,Ys), num(R), list(num,Zs),
        size(ub,length,Zs,length(Ys)+length(Xs)) )
   + cost(ub,steps,
          length(Xs)**2+
          length(Xs)*log(2,int(R))+
          length(Xs)*length(Ys)+
          2.0*length(Xs)+
          1).
```

Listing 5.4: Analysis Result for program in Listing 5.3.

In Listing 5.4, we show the results of analyzing this program. As we can see, the expression obtained by the analysis and the one obtained mathematically are equivalent.

5.4. Other Improvements

Since the CiaoPP cost analysis depends on a number of other analyses present in the system (e.g., modes, types/shapes, non-failure, or determinacy/mutual exclusion), which are also in continuous development, some maintenance work is required in order to keep all these analyses interact and exchange information properly, e.g., adapting to changes in the representation of information. We have performed such work during the development of this thesis, additionally improving the way such information can be extracted and used. This includes some maintenance and improvement work on the reading and treatment of cost-related "trust" assertions, the correct use of non-failure information at the program point level, or performing additional simplifications to the cost functions used.

Chapter 6

Experimental Results

Since its inception during the development of my Bachelor thesis, `ciao_tezos` has extended its capabilities, including new functionalities to make the analysis of Tezos smart contracts more precise. Currently, our Edo cost model includes 164 assertions, covering most Michelson instructions, a great improvement over the 97 assertions present in the cost model that we had in November and a proof of the extensibility and flexibility capabilities of our system.

Regarding the Michelson to CHC IR translator, it is 1500 lines long, of which 480 lines correspond to instructions definitions, transliterated from the specification, and 230 to instruction metadata.

Our framework can currently infer *storage* consumption and the gas cost associated to the *execution* of a program, but it could be extended to measure other costs, such as those derived from the *type checking* of Michelson contracts. Also, the precision of the storage measurement could be greatly improved using the *sized types-based* analysis previously mentioned in Section 4.2.1.

We have tested this prototype on a wide range of contracts, a few self-made and most of them published, both in Michelson's "*A contract a day*" examples and the Tezos blockchain itself. Experimental results for a selection of them are listed in Table 6.1. In this selection, we have tried to cover a reasonable range of Michelson data structures and control-flow instructions, as well as different cost functions using different metrics.

Column **Contract** lists the names of the contracts analyzed, and **Metrics** shows the metrics used to measure the parameter and the storage. The metrics used are: *value* for the numeric value of an integer, *length* for the length of a list, and *size* which maps every ground term to the number of constants and functions appearing in it. Column **Resource A**(nalysis) shows the complexity order of the resource usage function inferred by the analysis in terms of the sizes of the parameter (α) and the storage (β), including the values 1 if the inferred function is constant, ∞ if it is infinite and $\top$ it no safe bounds could be inferred. However, the actual cost functions inferred by our analysis also include the constants. For complex metrics derived from tuples, subindices starting from *1* are used to refer to the size of each argument; e.g., α_2 refers to the size of the second argument of the parameter. An additional token is included in the **Parameter** column, stating to which entrypoint it corresponds in case the contract has more than one entrypoint. Finally, **Time** shows the time (in millisec-

Table 6.1: Results of analysis for selected Michelson contracts.

Contract	Metrics		Resource A.		Time
	Parameter (α)	Storage (β)	*gas*	*storage*	(ms)
`reverse`	*length*	*length*	α	$\alpha - \beta$	372
`addition`	*value*	*value*	$\log \alpha$	$\log \frac{\alpha}{\beta}$	255
`michelson_arith`	*value*	*value*	$\log(\alpha^2 + \beta)$	$\log(\alpha^2 + \beta)$	248
`bytes`	*value*	*length*	β	$\top$	302
`list_inc`	*value*	*length*	β	0	431
`lambda`	*value*	*value*	$\log \alpha$	$\log \beta$	253
`lambda_apply`	*size*	*value*	∞	∞	222
`inline`	*size*	*value*	$\log \beta$	$\log \beta$	906
`cross_product`	$(length, length)$	*value*	$\alpha_1 + \alpha_2$	$\top$	397
`lineal`	*value*	*value*	α	$\top$	534
`assertion_map`	$(value, size)$	*length*	$\log \beta * \log \alpha_1$	$\top$	412
`quadratic`	*length*	*length*	$\alpha * \beta$	$\top$	508
`queue`	*size*	$(value, size, length)$	$\log \beta_1 * \log \beta_3$	$\top$	1108
`king_of_tez`	*size*	$(value, value, size)$	1	$\top$	846
`set_management`	*length*	*length*	$\alpha * \log \beta$	α	386
`max_list`	*length*	*size*	α	$\top$	684
`zipper`	*length*	$(length, length, length)$	1	$\top$	1032
`auction`	*size*	$(value, value, size)$	1	$\top$	854
`union`	$(length, length)$	*length*	$\alpha_1 * \log \alpha_2$	$\alpha_1 + \alpha_2 + \beta$	336
`append`	$(length, length)$	*length*	α_1	$\alpha_1 + \alpha_2 + \beta$	723
`subset`	$(length, length)$	*size*	$\alpha_1 * \log \alpha_2$	0	513
`list_api`	`(cons)` *value*	*length*	1	1	1116
	`(map)` *size*		∞	0	
	`(reverse)` *size*		β	0	
	`(default)` *size*		1	β	
`arithmetic_api`	`(add)` *value*	*value*	$\log(\alpha + \beta)$	$\log \alpha$	389
	`(sub)` *value*		$\log(\alpha + \beta)$	$\log \alpha$	
	`(default)` *size*		1	$\log \beta$	
`dispatcher`	`(do)` *size*	*size*	∞	0	684
	`(default)` *size*		1	0	
`ops_dispatcher`	*value*	$(value, size)$	$\log \beta_1$	$\top$	928

onds) taken to perform all the analyses using the different abstract domains provided by `CiaoPP`, version 1.20 on a medium-loaded 2.3 GHz Dual-Core Intel Core i5, 16 GB of memory, running macOS Big Sur 11.4. Many optimizations and improvements are possible, as well as more comprehensive benchmarking, but we believe that the results shown suggest that relevant bounds can be obtained in reasonable times, which, given the relative simplicity of development of the tool, seem to support our expectations regarding the advantages of our approach.

Chapter 7

Conclusions and Future Work

During the development of this Master's thesis, we have extended the capabilities of the `ciao_tezos` tool, which is now able to analyze more complex contracts and infer safe bounds for another important Tezos resource (besides *gas*): *storage*. Thanks to the Parametric Cost Analysis approach, we were able to support each new Tezos protocol iteration in a matter of hours, by simply modifying our cost model and—when needed—the translation tool. Also, the improvements made to the `CiaoPP` framework to deal with the problems posed by Michelson smart contracts might be useful in the future to analyze programs written in other languages. The application of partial evaluation techniques in the development of this tool has made the inclusion of new features, such as entrypoints, quite simple. The obtained results were quite promising, as they were obtained in a reasonable time by running the `CiaoPP` analyses on the unaltered output of the translator, and have drawn the attention of Tezos, with whom we are collaborating in the development of our Michelson assertion language.

In general, our new experience and promising results show the feasibility of the approach that we propose, allowing rapid, flexible, and effective development of cost analyses for smart contracts, which can be specially useful in the rapidly changing environment in blockchain technologies, where new languages arise frequently and cost models are modified with each platform iteration.

Despite the positive results of our (experimental) assessment, there are particular Michelson constructs which our tool cannot deal with as expected. In order to overcome these problems, we may have to apply more modern analyses, such as *sized types* or develop further improvements to our built-in recurrence relations solver.

As a final, general conclusion, this thesis gives more support to our hypothesis that our approach, based on Parametric Cost Analysis, provides a quick development path for cost analyses for new smart contract platforms and languages, or easily adapting existing ones to changes.

Bibliography

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

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger." 2016.

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

[4] C. W. Enumeration, "Cwe-400: Uncontrolled resource consumption," 2021. Accessed: 2021-05-04.

[5] S. C. W. Classification and T. Cases, "Swc-128." `https://swcregistry.io/docs/SWC-128`, 2018. Accessed: 2021-05-04.

[6] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "Running on fumes - preventing out-of-gas vulnerabilities in ethereum smart contracts using static resource analysis," in *VECoS 2019*, vol. 11847 of *LNCS*, pp. 63–78, Springer, October 2019.

[7] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: gas analysis and optimization for ethereum smart contracts," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020*, vol. 12079 of *LNCS*, pp. 118–125, Springer, 2020.

[8] 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.

[9] J. Navas, E. Mera, P. Lopez-Garcia, and M. Hermenegildo, "User-Definable Resource Bounds Analysis for Logic Programs," in *Proc. of ICLP'07*, vol. 4670 of *LNCS*, pp. 348–363, Springer, 2007.

[10] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo, "User-Definable Resource Usage Bounds Analysis for Java Bytecode," in *BYTECODE'09*, vol. 253 of *ENTCS*, pp. 6–86, Elsevier, March 2009.

[11] A. Serrano, P. Lopez-Garcia, and M. V. Hermenegildo, "Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types," *TPLP, ICLP'14 Special Issue*, vol. 14, no. 4-5, pp. 739–754, 2014.

[12] 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.*

WS on Automated Debugging–AADEBUG, pp. 155–170, U. Linköping Press, May 1997.

[13] 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*, pp. 161–192, Springer-Verlag, 1999.

[14] G. Puebla, F. Bueno, and M. V. Hermenegildo, "Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs," in *Proc. of LOPSTR'99*, LNCS 1817, pp. 273–292, Springer-Verlag, March 2000.

[15] M. Hermenegildo, G. Puebla, F. Bueno, and P. L. Garcia, "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)," *Science of Computer Programming*, vol. 58, no. 1–2, pp. 115–140, 2005.

[16] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo, "Incremental and Modular Context-sensitive Analysis," *TPLP*, vol. 21, pp. 196–243, January 2021.

[17] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *ACM Symposium on Principles of Programming Languages (POPL'77)*, pp. 238–252, ACM Press, 1977.

[18] M. Méndez-Lojo, J. Navas, and M. Hermenegildo, "A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs," in *LOPSTR*, vol. 4915 of *LNCS*, pp. 154–168, Springer-Verlag, August 2007.

[19] 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.

[20] E. Mera, P. Lopez-Garcia, M. Carro, and M. V. Hermenegildo, "Towards Execution Time Estimation in Abstract Machine-Based Languages," in *PPDP'08*, pp. 174–184, ACM Press, July 2008.

[21] 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 *Proceedings of LOPSTR'13*, vol. 8901 of *LNCS*, pp. 72–90, Springer, 2014.

[22] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder, "Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR," in *Proc. of FOPARA*, vol. 9964 of *LNCS*, pp. 81–100, Springer, 2016.

[23] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo, "Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks," in *Logic-Based Program Synthesis and Transformation - 27th International Symposium*, vol. 10855 of *LNCS*, Springer, 2018.

[24] 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.

[25] J. Peralta, J. Gallagher, and H. Sağlam, "Analysis of imperative programs through analysis of constraint logic programs," in *Static Analysis. 5th International Symposium, SAS'98, Pisa* (G. Levi, ed.), vol. 1503 of *LNCS*, pp. 246–261, 1998.

[26] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017*, pp. 442–446, IEEE Computer Society, February 2017.

[27] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina, "Computing exact worst-case gas consumption for smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*, vol. 11247 of *LNCS*, pp. 450–465, Springer, November 2018.

[28] B. Wegbreit, "Mechanical Program Analysis," *Communications of the ACM*, vol. 18, pp. 528–539, September 1975.

[29] 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.

[30] 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.

[31] 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.

[32] P. Lopez-Garcia, M. Klemen, U. Liqat, and M. V. Hermenegildo, "A General Framework for Static Profiling of Parametric Resource Usage," *TPLP (ICLP'16 Special Issue)*, vol. 16, no. 5-6, pp. 849–865, 2016.

[33] M. Klemen, N. Stulova, P. Lopez-Garcia, J. F. Morales, and M. V. Hermenegildo, "Static Performance Guarantees for Programs with Run-time Checks," in *Int'l. Symp. on Principles and Practice of Declarative Programming (PPDP'18)*, ACM, September 2018.

[34] M. Klemen, P. Lopez-Garcia, J. Gallagher, J. Morales, and M. V. Hermenegildo, "A General Framework for Static Cost Analysis of Parallel Logic Programs," in *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*, vol. 12042 of *LNCS*, pp. 19–35, Springer-Verlag, April 2020.

[35] P. Vasconcelos and K. Hammond, "Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs," in *IFL'03*, vol. 3145 of *LNCS*, pp. 86–101, Springer, 2003.

[36] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," *ACM TOPLAS*, vol. 34, no. 3, pp. 14:1–14:62, 2012.

[37] B. Grobauer, "Cost recurrences for DML programs," in *Proceedings of ICFP '01*, (New York, NY, USA), pp. 253–264, ACM, 2001.

[38] A. Igarashi and N. Kobayashi, "Resource usage analysis," in *Symposium on Principles of Programming Languages*, pp. 331–342, ACM, 2002.

[39] F. Nielson, H. Nielson, and H. Seidl, "Automatic complexity analysis," in *Programming Languages and Systems*, LNCS, pp. 243–261, Springer, 2002.

[40] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs, "Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs," in *Proceedings of PPDP'12*, pp. 1–12, ACM, 2012.

[41] E. Albert, S. Genaim, and A. N. Masud, "More Precise yet Widely Applicable Cost Analysis," in *Proc. of VMCAI'11*, vol. 6538 of *LNCS*, pp. 38–53, Springer, 2011.

[42] S. Gulwani, K. K. Mehra, and T. M. Chilimbi, "SPEED: Precise and Efficient Static Estimation of Program Computational Complexity," in *The 36th Symposium on Principles of Programming Languages (POPL'09)*, pp. 127–139, ACM, 2009.

[43] A. O. Maroneze, S. Blazy, D. Pichardie, and I. Puaut, "A formally verified WCET estimation tool," in *Workshop on Worst-Case Execution Time Analysis – WCET 2014*, vol. 39 of *OASICS*, pp. 11–20, Schloss Dagstuhl, 2014.

[44] M. Hofmann and G. Moser, "Multivariate amortised resource analysis for term rewrite systems," in *13th International Conference on Typed Lambda Calculi and Applications* (T. Altenkirch, ed.), vol. 38 of *LIPIcs*, pp. 241–256, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, July 2015.

[45] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann, "Relational cost analysis," in *Principles of Programming Languages, POPL 2017* (G. Castagna and A. D. Gordon, eds.), pp. 316–329, ACM, 2017.

[46] M. Avanzini and U. D. Lago, "Automating sized-type inference for complexity analysis," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 43:1–43:29, 2017.

[47] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation," in *European Symposium on Research in Computer Security – ESORICS 2017*, vol. 10492 of *Lecture Notes in Computer Science*, pp. 260–277, Springer, September 2017.

[48] W. Qu, M. Gaboardi, and D. Garg, "Relational cost analysis for functional-imperative programs," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, pp. 92:1–92:29, 2019.

[49] Z. Kincaid, J. Breck, J. Cyphert, and T. W. Reps, "Closed forms for numerical loops," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 55:1–55:29, 2019.

[50] G. Moser and M. Schneckenreither, "Automated amortised resource analysis for term rewrite systems," *Sci. Comput. Program.*, vol. 185, 2020.

[51] M. A. T. Handley, N. Vazou, and G. Hutton, "Liquidate your assets: reasoning about resource usage in liquid haskell," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 24:1–24:27, 2020.

[52] V. Pérez Carrasco, "Analysis of smart contracts using horn clauses," Master's thesis, Universidad Politécnica de Madrid, ETSII, E-28660, Boadilla del Monte, Madrid, Spain, June 2020. BSc Thesis.

[53] 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," *TPLP*, vol. 12, no. 1–2, pp. 219–252, 2012.

[54] 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.

[55] A. Serrano, P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo, "Sized Type Analysis for Logic Programs," in *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement (technical communication)* (T. Swift and E. Lamma, eds.), vol. 13, pp. 1–14, Cambridge U. Press, August 2013.

[56] F. Bueno, D. Cabeza, M. V. Hermenegildo, and G. Puebla, "Global Analysis of Standard Prolog Programs," in *ESOP*, 1996.

[57] G. Puebla, F. Bueno, and M. V. Hermenegildo, "An Assertion Language for Constraint Logic Programs," in *Analysis and Visualization Tools for Constraint Programming*, no. 1870 in LNCS, pp. 23–61, Springer-Verlag, 2000.

Appendix A

CHC IR Representation of a Contract with Entrypoints

```
:- module(A, [], [ciao_tezos(cost_models/edo)]).

:- use_module(ciao_tezos(cost_models/michelson_preds)).

:- entry cons(A, B, C, D)
   : ( mutez(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) ).

cons(A, B, [], [A|B]) :-
    '$unpair', '$if_left', '$if_left', '$if_left',
    cons(A, B, [A|B]),
    nil([]),
    '$cons_pair',
    '$storage_diff_list'(B, [A|B], 8).

:- entry map(A, B, C, D)
   : ( lambda(A), list(mutez, B), var(C), var(D),
       ground(A), ground(B), mshare([C,D], [[C], [D]]) ).

map(A, B, [], C) :-
    '$unpair', '$if_left', '$if_left', '$if_left',
    '$swap',
    '$list_map'(B),
    map__0(B, C, A, D),
    '$dip',
    '$drop'(D),
    nil([]),
    '$cons_pair',
    '$storage_diff_list'(B, C, 8).

:- impl_defined(lambda_0/2).

:- trust pred lambda_0(A, B)
```

```
    : ( mutez(A), var(B), ground(A) )
    => ( mutez(A), mutez(B), ground(A), ground(B) )
    + ( not_fails, is_det,
        cost(ub, michelson_allocations, inf),
        cost(ub, michelson_bytes_read, inf),
        cost(ub, michelson_bytes_written, inf),
        cost(ub, michelson_gas, inf),
        cost(ub, michelson_reads, inf),
        cost(ub, michelson_steps, inf),
        cost(ub, michelson_writes, inf)).

dispatcher_0(A, B, C) :-
    lambda_0(B, C).

map__0([], [], A, A).
map__0([A|B], [C|D], E, F) :-
    '$dip', '$dup', '$exec',
    dispatcher_0(E, A, C),
    map__0(B, D, E, F).

:- entry reverse(A, B, C, D)
    : ( unit(A), list(mutez, B), var(C), var(D),
        ground(A), ground(B), mshare([C,D], [[C], [D]]) ).

reverse('()', A, [], B) :-
    '$unpair', '$if_left', '$if_left',
    '$drop'('()'),
    nil([]),
    '$swap',
    '$list_iter'(A),
    iter__1(A, [], B),
    nil([]),
    '$cons_pair',
    '$storage_diff_list'(A, B, 8).

iter__1([], A, A).
iter__1([A|B], C, D) :-
    cons(A, C, [A|C]),
    iter__1(B, [A|C], D).

:- entry default(A, B, C, D)
    : ( unit(A), list(mutez, B), var(C), var(D),
        ground(A), ground(B), mshare([C,D], [[C], [D]]) ).

default('()', A, [], []) :-
    '$unpair', '$if_left',
    '$dropn'(2, ['()',A]),
    nil([]),
    nil([]),
```

```
83      '$cons_pair',
84      '$storage_diff_list'(A, [], 8).
```

Listing A.1: CHC IR representation of the contract in Listing 4.5.