Cost Analysis of Smart Contracts via Parametric Resource Analysis*

Víctor Pérez[1], Maximiliano Klemen[1], Pedro López-García[1,3], José Francisco Morales[1], and Manuel Hermenegildo[1,2]

[1] IMDEA Software Institute
{victor.perez,maximiliano.klemen,pedro.lopez,
josef.morales,manuel.hermenegildo}@imdea.org
[2] Universidad Politécnica de Madrid (UPM)
[3] Spanish Council for Scientific Research (CSIC)

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 bounds in smart contracts. The approach is based on *Parametric Resource 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. We argue that this approach offers a rapid, flexible, and effective method for the development of cost analyses for smart contracts.

Keywords: Blockchain · Smart Contracts · Parametric Resource Analysis · Static Analysis · Constraint Horn Clauses · Program Transformation

1 Introduction

Due to the nature of blockchain platforms [63, 6], smart contracts [60] and their storage are replicated in every node running the chain, and any call to a contract is executed on every client. This fact has led many smart contract platforms to include upper bounds on execution time and storage, as well as fees associated with running a contract or increasing its storage size. More concretely, in order

* Partially funded by MICINN PID2019-108528RB-C21 *ProCode* and Madrid P2018/TCS-4339 *BLOQUES-CM*. Thanks to Vincent Botbol, Mehdi Bouaziz, and Raphael Cauderlier from Nomadic Labs, and Patrick Cousot, for their comments.

to limit execution time, smart contract platforms make use of a concept called "gas," so that each instruction of the smart contract language usually has an associated cost in terms of this resource. If a transaction exceeds its allowed *gas* consumption, its execution is stopped and its effects reverted. However, even if a transaction does not succeed because of *gas* exhaustion, it is included in the blockchain and the fees are taken. Similarly, there are limitations and costs related to *storage size*. The cost of running a contract can then be expressed in terms of these two resources, *gas* consumed and *storage*.

In this context, knowing the cost of running a contract beforehand can be useful, since it allows users to know how much they will be charged for the transaction, and whether *gas* limits will be exceeded or not. However, this is not straightforward in general. Many smart contract platforms do provide users with simulators which allow performing dry runs of smart contracts in their own node before performing actual transactions. But this of course returns cost data only for specific input values, and provides no hard guarantees on the costs that may result from processing the arbitrary inputs that the contract may receive upon deployment. 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.

Thus, formal verification of smart contracts, and in particular analysis and verification of their resource consumption, is receiving increased attention. At the same time, many different blockchain platforms now exist, using different languages and cost models, which often take into account different resources and count them in different, platform-specific ways. Furthermore, within each platform, the models can also evolve over time. As a consequence, the few existing resource analysis tools for smart contracts, such as GASTAP [5], GASOL [4], or MadMax [22], tend to be quite specific, focusing on just a single platform or language, or on small variations thereof.[4] This makes approaches that would allow quick development of new cost analyses or easily adapting existing ones potentially attractive in this context.

Parametric Resource Analysis (also referred to as user-defined resource analysis) [52, 51, 59] is an approach that simplifies the implementation of analyzers that infer safe functional bounds on different related resources and with different resource consumption models. Our objective in this paper is to explore the application of this general approach to the rapid and effective development of static analyses for gas consumption in smart contracts. To this end, we use the implementation of the method in the `CiaoPP` [29] framework, and apply it to the Tezos platform [6] and its Michelson language [1] as a proof of concept.

In the rest of the paper we start by providing an overview of the general approach (Sect. 2), and then we illustrate successively the translation process (Sect. 3), how the cost model is encoded (Sect. 4), and how the analysis is performed (Sect. 5), first in general and then applied to the Michelson language. We also provide some experimental results in Sect. 6. Sect. 7 then discusses other related work and Sect. 8 presents our conclusions and future work.

[4] We discuss this and other relevant related work further in Section 7.

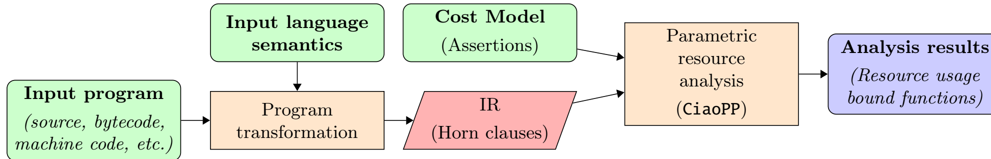

Fig. 1. Overview of the Parametric Resource Analysis approach.

2 The Parametric Resource Analysis Approach

We start by providing an overview of the approach (Fig. 1). Before getting into the resource analysis itself, a basic technique used in the model, in order to support different input languages, is to translate input programs to a Horn clause-based intermediate representation [46], that we refer to as the "CHC IR," a technique used nowadays in many analysis and verification tools [54, 27, 46, 20, 21, 25, 13, 35, 18]. The CHC IR is handled uniformly by the analyzers, and the results are then reflected back to the input language. To perform the Parametric Resource Analysis, assertions are used to define the resources of interest, including compound resources, and the consumption that basic elements of the input language (e.g., commands, instructions, bytecodes, built-ins, etc.) make of such resources. This constitutes the *cost model*. This model is normally generated once for each input language, and is the part modified if the costs change or different resources need to be inferred. Given an input program and the cost model, the parametric analyzer then infers, for each program point (block, procedure, etc.), safe resource usage bound *functions* that depend on data sizes and possibly other parameters. Both the resource consumption expressions inferred and those appearing in the cost models can include e.g., polynomial, summation, exponential, and logarithmic, as well as multi-variable functions. This overall approach, pioneered and supported by the `CiaoPP` framework, has been successfully applied to the analysis, verification, and optimization of resource consumption for languages ranging from source to machine code, and resources ranging from execution time to energy consumption [50, 47, 51, 41, 40, 39, 42].

3 Translating into the CHC IR

As mentioned above, in order to support different programming languages and program representations at different compilation levels, each input language is translated into a Horn clause-based intermediate program representation, the CHC IR [46]. A (Constrained) Horn clause ((C)HC) is a formula of first-order predicate logic (generalized with constraints) of the form $\forall(S_1 \wedge \ldots \wedge S_n \rightarrow S_0)$ where all variables in the clause are universally quantified over the whole formula, and $S_0, S_1, \ldots, S_n$ are atomic formulas, also called literals, or constraints. CHCs are usually written: S_0 `:-` $S_1, \ldots, S_n$, where S_0 is referred to as the *head* and $S_1, \ldots, S_n$ as the *body*. Given a program p in an input language L_p, plus a definition of the semantics of L_p, the objective is to translate p into a set of Horn clauses that capture the semantics of p. Two main styles are generally used for encoding the operational semantics of L_p [18]: small-step (structural operational

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

Listing 1.1. A Michelson contract that reverses a list.

semantics) [55], as in [54], or big-step (natural semantics) [34], as in [46]. We will be concerned herein with the latter, among other reasons because the big-step approach is very direct for the case of a language that is structured and defined functionally, such as Michelson.

Typically, a CHC interpreter of L_p, I, in one of the styles above, together with a term-based representation of p and its store, would suffice to reflect the program semantics. However, precise analyses often require a tighter correspondence between predicates and body literals in the CHCs and the *blocks* (e.g., in a control-flow graph) and statements (e.g., *calls* and *built-ins*) for p. For example, for an imperative program, the CHCs typically encode a set of connected code *blocks*, so that each block is represented by a CHC: $\langle block_id \rangle(\langle params \rangle)$ `:-` S_1, ... ,S_n. The *head* represents the entry point to the block and its parameters, and the *body* the sequence of steps in the block. Each of these S_i steps (or *literals*) is either a *call* to another (or the same) block or a call to one of the basic operations implemented by the interpreter I. Thus, depending on the input language, literals can represent bytecode instructions, machine instructions, calls to built-ins, constraints, compiler IR instructions, etc.

Techniques such as partial evaluation and program specialization offer powerful methods to obtain such translations. In particular, using the first Futamura projection [17], I can be specialized for a given input program p, which, with appropriate simplifications, results in a set of predicates with the desired correspondences. A direct, automatic translator can be obtained by specializing a CHC partial evaluator for I (second Futamura projection), which can then be applied to any program p. In general, these transformations may be automatic, manual, or use a combination of techniques. Also, preliminary transformations may be required to express the semantics at the right abstraction level, e.g., making all variable scoping explicit, using Static Single Assignment (SSA), reducing control constructs, etc. [46].

The Michelson Language and its Semantics. Michelson is the "native" language used by the Tezos platform. It is interpreted, strongly-typed, and stack-based. Despite being a low-level language, Michelson provides some high-level data structures such as lists, sets, maps, and arbitrary precision integers.

Michelson contracts consist of three sections. The *parameter* and *storage* sections stipulate the types of the input argument and the storage. E.g., in Listing 1.1 both are described as lists of Michelson integers. The *code* section contains the sequence of instructions to be executed by the Michelson interpreter. This interpreter can be seen as a pure function that receives a stack and returns a result stack without altering its environment. The input stack contains just a pair consisting of the *parameter* and the contract *storage*. The output stack will contain just a pair consisting of the *list of blockchain operations* to be executed

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

Fig. 2. Semantics of some Michelson instructions.

after the contract returns and the *updated storage*, to be used as storage value in the following call to the contract. I.e.:

$$Interpreter\colon\ (pair\ parameter\ storage) : [] \to (pair\ (list\ operation)\ storage) : []$$
$$(p, s) : [] \mapsto (l, s') : []$$

The Michelson instructions can also be seen as pure functions receiving an input stack and returning a result stack. Fig. 2 shows the semantics of the Michelson instructions used in Listing 1.1—overall, there are 116 typed instructions and 23 macros. Continuing with the example, its purpose is to reverse the list passed as a parameter and store it. First, `CAR` discards the storage of the contract, as only the list passed as parameter is needed for the computation. Then, the `NIL` instruction inserts an empty list on top of the stack. The type of the elements that will fill the resulting list needs to be provided, in this case integers. `SWAP` simply exchanges the top two elements of the stack. After running these instructions, the stack will have the following shape: $parameter : ([]) : []$.

The interpreter will now iterate over the input list, prepending each of its elements to the new list and reversing the former in the process. This action is carried out by the `ITER` instruction, which traverses the elements of a list, performing the action indicated by its argument: a macro or a sequence of instructions; in our case, just `{ CONS }`. `CONS` receives a stack whose top is an element and a list of the same type, and returns a stack with just the list on top, but where the list has the element preprended, while the rest of the stack is unchanged. Taking into account the semantics of `CONS`, the semantics of the loop within the contract can be defined as:

$$l_a : l_b : S \mapsto \texttt{ITER}(l_a : l_b : S) = \begin{cases} l_b : S & \text{if } l_a = [] \\ \texttt{ITER}(l'_a : (el : l_b) : S) & \text{if } l_a = el : l'_a \end{cases}$$

There are other instructions which receive code as an argument: the control structures in the language, e.g., `IF` or `LOOP`, are instructions which receive one or two *blocks* of code. Likewise, other instructions receive other kinds of arguments, such as `NIL`, which as we saw receives the type of the list to build; or `PUSH`, which receives the type and value of the element to place on top of the stack. Once

car([(A, _)|S], [A|S]).	nil(_, S, [[]|S]).
swap([A, B|S], [B, A|S]).	pair([A, B|S], [(A, B)|S]).
iter(Body, [L|S0], S1) ← iter(L, Body, S0, S1).	cons([X, Xs|S], [[X|Xs]|S]).
iter([], _, S, S). iter([X|Xs], Body, S0, S2) ← run(Body, [X|S0], S1), iter(Xs, Body, S1, S2).	

Fig. 3. Semantics of the instructions of Fig. 2 in CHC.

the list has been reversed, the contract inserts a list of operations on top of the stack, via the NIL instruction, and builds a pair from the two elements left in the stack, using the PAIR instruction. This way, the result stack will have the required type, i.e., length and type of its elements:

$$(pair\ (list\ operation)\ storage) : [], \quad \text{where} \quad storage \equiv (list\ int)$$

As a concrete example, a call to this contract with the list of numbers from 1 to 3 as parameter would present the following input (S_0) and output (S_1) stacks:

$$S_0 = ((1 : 2 : 3), _) : [] \mapsto S_1 = ([], (3 : 2 : 1)) : []$$

Note that, as the first instruction in the contract discards the storage, its value is irrelevant to obtain the result of the computation.

As mentioned before, in addition to performing operations over terms in the stack, Michelson instructions can also return *external operations* (i.e., instructions that perform actions at the blockchain level) to be added to the list of operations in the return stack. Lack of space prevents us from going into details, but these operations can be: *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).

CHC encoding. We implement the Michelson semantics as a big-step recursive interpreter, via a direct transliteration of the semantics into CHCs (using the Ciao system [28]). Fig. 3, shows the CHC encoding of the instructions of Fig. 2. Data structures are represented in the usual way with Herbrand terms.[5] The interpreter in turn is encoded by the following clauses:[6]

```
run([], S0, S) :- S=S0.
run([Ins|Insns], S0, S) :- ins(Ins, S0, S1), run(Insns, S1, S).
% Dispatcher (one clause for each I/n instruction)
ins(<<I>>(A1,...,An), S0, S) :- <<I>>(A1,...,An,S0,S).
```

Predicate run/3 takes the input program and the initial stack (S0), and reduces it by executing the sequence of Michelson instructions to obtain the resulting stack S1. ins/3 is the instruction dispatcher, which connects each instruction term (e.g., push(X)) with its CHC definition (e.g., push(X,S0,S)) (see Fig. 3).

[5] We do not include the types in Fig. 3 for brevity; they will be present however in the cost model assertions of Section 4.

[6] In the actual code, state variables are made implicit by using Definite Clause Grammar (DCG) syntax. We have left all variables explicit however for clarity.

The Michelson to CHC IR Translation. We derive a simple translator, based on a specialization of a CHC partial evaluation algorithm for this particular recursive interpreter. In this process special care is taken to materialize stack prefixes as actual predicate arguments.

Preliminary transformations. As preliminary transformations we introduce *labeled* blocks for sequences of instructions in the program, to help in later steps of partial evaluation. For the sake of clarity, we consider them simply as new predicate definitions (we obviate for conciseness some additional information needed to trace back blocks to the original program points). We also rely on a simple implementation within the system of Michelson type checking, which makes knowing the type of the stack (and thus of the operands) at each program point a decidable problem. This allows us to specialize polymorphic instructions, depending on the type of the passed arguments. This is particularly useful to specify (as we will see later) the semantics and cost of each instruction variant, which can be vary depending on those static types. E.g., the `ADD` instruction is translated into one of seven possible primitive operations, depending on the type of the addends:

$$ADD[A,B] \rightarrow \begin{cases} \texttt{add_intint} & \text{if } int(A), int(B) \\ \texttt{add_intnat} & \text{if } int(A), nat(B) \\ \texttt{add_natint} & \text{if } nat(A), int(B) \\ \texttt{add_natnat} & \text{if } nat(A), nat(B) \\ \texttt{add_timestamp_to_seconds} & \text{if } timestamp(A), int(B) \\ \texttt{add_seconds_to_timestamp} & \text{if } int(A), timestamp(B) \\ \texttt{add_tez} & \text{if } mutez(A), mutez(B) \end{cases} \tag{1}$$

Translation using partial evaluation. Based on our interpreter, we derive stepwise a simple translator which combines a hand-written specializer for the `run/3` predicate, a stack deforestation pass (including each stack element instead of the stack itself as predicate arguments), and a generic partial evaluation for the primitive instruction definitions (e.g., evaluate conditions, arithmetic instructions, etc.). Michelson control-flow instructions receive both the control condition and the code to execute as inputs, e.g.:

```
if(Bt,Bf,[B|S0],S) :- '$if', if_(B,Bt,Bf,S0,S).
if_(true, Bt,_Bf,S0,S) :- run(Bt,S0,S).
if_(false,_Bt,Bf,S0,S) :- run(Bf,S0,S).
```

By construction, the code arguments are bound, as explained in the preliminary transformations, to new constants representing code blocks dispatched from `ins/3`. For each call, partial evaluation will unfold `if(Bt,Bf,S0,S2)` as `'$if', S0=[B|S1], if__0(B,S1,S2)` and generate new instances, e.g.:

```
if__0(true, S0,S) :- ... % unfolded run(<<Bt>>,S0,S).
if__0(false,S0,S) :- ... % unfolded run(<<Bf>>,S0,S).
```

The stack deforestation step is specially useful in the output of control-flow instructions, which receive $n+m$ arguments instead of the lists of variables, where n is the size of the input stack and m of the output stack. This transformation

```
parameter (pair int (list int)) ;
storage int ;
code { CAR ;
       UNPAIR ;
       DUP ;
       SUB ;
       DIIP { PUSH int 0 } ;
       IFNEQ { ITER { ADD } } { DROP } ;
       NIL operation ;
       PAIR }
```

Listing 1.2. A Michelson contract suitable for partial evaluation.

is possible thanks to Michelson's semantics, which forbids changes to the type of the stack in loops and forces the type of both output stacks in branch instructions to match. E.g., for the simple branch instruction `IF`:

```
if__0(true, I0,I1,...,In,O0,O1,...,Om) :- ...
if__0(false,I0,I1,...,In,O0,O1,...,Om) :- ...
```

Following the idea of abstracting away the stack, the translation also abstracts away simple data structures, such as pairs, whenever possible.

Cost-preserving encoding. In order to precisely capture the actual cost of instructions, while allowing aggressive program transformations such as unfolding, partial evaluation, and replacing the stack arguments by actual parameters, the instruction definitions are extended to introduce *cost markers*, e.g.:

```
swap([A,B|S],[B,A|S]) :- '$swap'.
drop([X|S],S) :- '$drop'(X).
if(Bt,Bf,[B|S0],S) :- '$if', if_(B,Bt,Bf,S0,S).
if(true, Bt,_Bf,S0,S) :- run(Bt,S0,S).
if(false,_Bt,Bf,S0,S) :- run(Bf,S0,S).
```

Partial evaluation will replace each of the primitive operations (from a very reduced set) by its CHC definition in the output CHC IR, while the cost makers, whose main end is to keep a record of the consumed resources at each step, will be preserved. Note that as as a result of the transformations, some Michelson instructions that simply modify/access the stack will not even be represented in the output CHC IR, only their cost markers, if relevant.

Translation example. To illustrate all the steps described in this section, we show the resulting CHC representation for the contract shown in Listing 1.2. The direct translation of this contract can be found in Listing 1.3, whereas Listing 1.4 takes advantage of partial evaluation to perform significant, yet valid transformations, both in terms of semantics and resource semantics.

Another useful transformation performed by the translation is the inclusion of explicit arithmetic comparison operations in the contract. This way, Boolean conditions in control-flow predicates can be replaced by arithmetic tests, which not only makes the contract more readable for the human eye, but also easier to analyze. An example of this can be seen in Listing 1.5 and its CHC IR repre-

```
:- pred code/5 : int * list(int) * int * var * var.

code(A,B,C,D,E) :-
    '$car', '$dup', '$car', '$dip', '$cdr', '$dup',
    sub_intint(A,A,F),
    '$dip'(2), '$push'(0),
    neq(F,G),
    '$if',
    if__0(G,[B,0],[E]),
    nil(D),
    '$pair'.

if__0(true,[A,B],[C]) :-
    iter__1(A,[B],[C]).
if__0(false,[A,B],[B]) :-
    '$drop'(A).

iter__1([],[A],[A]) :-
    '$iter_end'.
iter__1([A|B],[C],[D]) :-
    '$iter',
    add_intint(A,C,E),
    iter__1(B,[E],[D]).
```

Listing 1.3. CHC IR representation of Listing 1.2.

```
:- pred code/5 : int * list(int) * int * var * var.

code(A,B,C,[],0) :-
    '$car', '$dup', '$car', '$dip', '$cdr', '$dup',
    sub_intint(A,A,0),
    '$dip'(2), '$push'(0),
    neq(0,false),
    '$if', '$drop'(B),
    nil([]),
    '$pair'.
```

Listing 1.4. CHC IR representation of Listing 1.2 with partial evaluation enabled.

sentation, Listing 1.6. In this contract one of the comparison operations and the evaluation of its result are performed in different predicates. This information can be encoded by attaching information about how they have been generated to the results of both `COMPARE` and `GT` instructions, which will propagate throughout the translation process inside the stack.

4 Defining Resources and Cost Models

After addressing in the previous section the parametricity of the approach w.r.t. the programming language, we now address parametricity w.r.t. resources and cost models. As mentioned before, the role of the *cost model* in parametric re-

```
parameter (pair int int) ;
storage int ;
code { UNPPAIIR ;
       DIIP { DUP } ;
       DUUUP ;
       SWAP ;
       CMPGT ;
       DIP CMPGT ;
       IF ASSERT FAIL ;
       NIL operation ;
       PAIR }
```

Listing 1.5. A Michelson contract with arithmetic comparisons.

```
:- pred code/5 : int * int * int * var * var.

code(A,B,C,[],D) :-
    '$dup', '$car', '$dip', '$cdr', '$dup', '$car', '$dip',
    '$cdr', '$dip'(2), '$dup', '$dip'(2), '$dup', '$dig'(3),
    '$swap',
    compare_int(A,C,E),
    gt(E,F),
    '$dip',
    compare_int(B,C,G),
    gt(G,H),
    '$if',
    if__0(A,C,B,C,H,C,D),
    nil([]),
    '$pair'.

if__0(A,B,C,D,E,F,G) :-
    A>B,
    '$if',
    if__1(C,D,F,G).
if__0(A,B,C,D,E,F,failed('()')) :-
    A=<B,
    '$push'('()'),failwith('()').

if__1(A,B,C,C) :-
    A>B.
if__1(A,B,C,failed('()')) :-
    A=<B,
    '$push'('()'),failwith('()').
```

Listing 1.6. CHC IR representation of Listing 1.5.

source analysis is to provide information about the resource consumption of the basic elements of the input language, which is then used by the analysis to infer the resource usage of higher-level entities of programs such as procedures, functions, loops, blocks, and the whole code. We start by describing a subset of the assertions proposed in [52] for describing such models, which are part of

the multi-purpose assertion language of the `Ciao`/`CiaoPP` framework [28, 56, 9], used in our experiments. First, the resources of interest have to be defined and given a name as follows:

```
:- resource ⟨resname⟩.
```

Then, we can express how each operation of the analyzed language affects the use of such resource, by means of assertions with `trust` status:

```
:- trust pred ⟨operation⟩ + cost(⟨approx⟩,⟨resname⟩,⟨arithexpr⟩).
```

where $\langle arithexpr \rangle$ expresses the resource usage as a function that depends on data sizes and possibly other parameters, and which, as mentioned before, can be polynomial, summation, exponential, or logarithmic, as well as multi-variable. The $\langle approx \rangle$ field states whether $\langle arithexpr \rangle$ is providing an upper bound (`ub`), a lower bound (`lb`), a "big O" expression, i.e., with only the order information (`oub`), or an Ω asymptotic lower bound (`olb`). Such assertions can also be used to describe the resource usage of builtins, libraries, external procedures (e.g., defined in another language), etc. Assertions can also include a *calls* field, preceded by `:`, stating properties that hold at call time. This allows writing several assertions for the same predicate to deal with polymorphic predicates whose resource semantics may differ depending on the call states. E.g., for `add` we can have assertions with call fields `int * int * var` and `flt * flt * var` with possibly different costs. An optional *success field*, preceded by `=>`, can also be used to state properties that hold for the arguments on success. Additionally, size metric information can be provided by users if needed using `size_metric`(Var,$\langle sz_metric \rangle$) properties, although in practice such metrics are generally derived automatically from the inferred types and shapes. These are the metrics used to measure data sizes, e.g.: list length, term depth, term size, actual value of a number, number of steps of the application of a type definition, etc. (see [52, 59] and the use therein of *sized types*). It is also possible to declare relationships between the data sizes of the inputs and outputs of procedures, as well as provide types and actual sizes (`size`(Var,$\langle approx \rangle$,$\langle sz_metric \rangle$,$\langle arithexpr \rangle$)). In addition to those presented, [52] proposes some additional mechanisms for defining other aspects of cost models, but they are not required for our presentation.

The Cost Model for the Tezos Platform. We now illustrate how to define the resources and cost model for our test case, the Tezos platform and its Michelson language, using the `Ciao` assertion language. The Tezos/Michelson cost model varies somewhat with each version of the protocol, which, as mentioned before, is one of the motivations for our approach. The model that we present has been derived from the OCaml source for the *Carthage* protocol. *Gas* is a *compound resource* that can be defined as a function of other *basic resources*:

$$gas(allocations, steps, reads, writes, bytes_read, bytes_written) =$$
$$= 2^{-7} * \begin{pmatrix} allocations \\ steps \\ reads \\ writes \\ bytes_read \\ bytes_written \end{pmatrix} \times \begin{pmatrix} 2 \\ 1 \\ 100 \\ 160 \\ 10 \\ 15 \end{pmatrix} \qquad (2)$$

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

Listing 1.7. 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 1.8. Assertions to declare *gas* as a compound resource.

In our cost model we first name the resources (Listing 1.7), and then define `michelson_gas` as a compound resource following Eq. 2 (Listing 1.8).

Each Michelson instruction will consume one or more of these basic resources, so the next step is to 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. This avoids having to provide information for all resources in the cost assertions for every instruction.[7]

We illustrate the process of declaring specific resource consumptions using the `ADD` instruction. Listing 1.9 shows the definition of this basic operation in the (OCaml) code of the Michelson interpreter, which contains not only the semantics of the instruction, but also its cost semantics. As mentioned before, this is a polymorphic instruction, so it may be transformed into different predicates in the translation process. In this case, we will focus on the instance dealing with integers, which was called `add_intint` in Eq. 1. Comparing Eq. 1 and List-

[7] We do not include examples of default assertions due to space constraints.

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

Listing 1.9. Some of the definitions for `ADD`.

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

Listing 1.10. Cost definition for `add_intint`.

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

Listing 1.11. `atomic_step_cost` definition.

ing 1.9 we can see that our translation process closely matches the Tezos internal representation of Michelson instructions.

The corresponding cost expression, as found in the Tezos source code, is shown in Listing 1.10, which is given in turn in terms of `atomic_step_cost`, Listing 1.11. This function is used to express the cost of a great number of operations, which, as in this case, can be given as a function of their arguments. Using this definition and that of `int_bytes`:

$$int_bytes(x) = 1 + \left\lfloor \frac{\log_2 |x|}{8} \right\rfloor \tag{3}$$

we can simplify `add_intint`'s cost expression:

$$\begin{aligned} cost_{add_intint}(A, B) &= 2 * \left(51 + \frac{\max\left(1 + \left\lfloor \frac{\log_2 |A|}{8} \right\rfloor, 1 + \left\lfloor \frac{\log_2 |B|}{8} \right\rfloor\right)}{62} \right) \\ &= 102 + \frac{1 + \left\lfloor \frac{\log_2 \max(|A|,|B|)}{8} \right\rfloor}{31} \end{aligned} \tag{4}$$

The assertion used to include this cost in our `CiaoPP` model is shown in Listing 1.12. It expresses the exact cost of this instruction in terms of its inputs. Both an upper and a lower bound are given. Since they are the same, the cost is exact—this can also be expressed with the `exact` keyword. Note that these assertions can also include properties of instruction arguments. In this case we state

```
:- 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_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 1.12. Cost assertion for `add_intint` in the cost model.

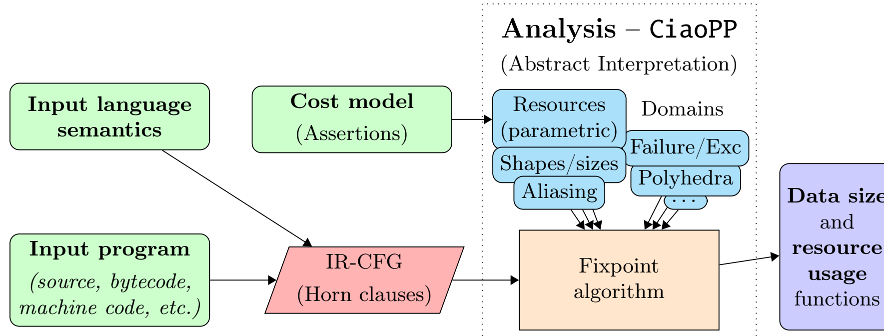

Fig. 4. Overview of analysis in the Parametric Resource Analysis approach.

the types and sizes of the arguments of the `add_intint` predicate on success, as well as other information such as non-failure, determinacy, or cardinality, which increase the precision of the resource analysis. In fact, since every Michelson instruction is a deterministic function defined in all of its domain, they never fail and they always return one solution. Note the direct correspondence between the arithmetic expression that defines the cost of the instruction and Eq. 4, which contributes to the readability of the model.

5 Performing the Resource Analysis

As already mentioned in Sect. 2, the input to the the parametric resource analyzer is the program in CHC IR form and the resource model (Fig. 1). The core analyzer is based on an approach in which recursive equations (cost relations), representing the resource consumption of the program, are extracted from the program and solved, obtaining upper- and lower-bound cost functions in terms of the program's inputs [62, 15, 16, 2, 59]. As mentioned before, these functions can be polynomial, exponential or logarithmic, etc., and they express the cost for each Horn clause (block) in the CHC IR, which can then be reflected back to the input language. Space restrictions prevent us from describing the process in detail; we provide an overview of the tasks performed by the analyzer (Fig. 4):

1. Perform all the required **supporting analyses**. This includes typically, among others: a) *sized types/shapes* analysis for inferring size metrics (for heap manipulating programs), to simplify the control-flow graph, and to improve overall precision (e.g., class hierarchy analysis); b) pointer *sharing/aliasing* analysis for correctness and precision; c) *Non-failure* (no exceptions) analysis, needed for inferring non-trivial lower bounds; d) *Determinacy* and *mutual exclusion* analyses to obtain tighter bounds; e) other instrumental analyses such as, e.g., *polyhedra* for handling constraints.
2. **Size analysis**: a) Set up recurrence equations representing the size of each (relevant) output argument as a function of input data sizes, based on data dependency graphs that determine the relative sizes of variable contents at different program points. The size metrics are derived from the inferred shape (type) information. Then, b) compute bounds to the solutions of these

```
:- pred code/4 : list(int) * list(int) * var * var.

code(A,B,[],C) :-
    '$car',
    nil([]),
    '$swap',
    iter__0(A,[],C),
    nil([]),
    '$pair'.

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

Listing 1.13. CHC IR representation of contract 1.1.

```
:- true pred code(A,B,C,D)
   : ( list(int,A), list(int,B), var(C), var(D) )
   => ( list(int,A), list(int,B), list(C), list(D),
        size(lb,A,length(A)), size(lb,B,length(B)),
        size(lb,C,0), size(lb,D,0) )
   +  ( cost(lb,michelson_gas,0.6875*length(A)+1.21875),
        cost(lb,michelson_steps,80*length(A)+140) ).

:- true pred code(A,B,C,D)
   : ( list(int,A), list(int,B), var(C), var(D) )
   => ( list(int,A), list(int,B), list(C), list(D),
        size(ub,A,length(A)), size(ub,B,length(B)),
       size(ub,C,inf), size(ub,D,inf) )
   + ( cost(ub,michelson_gas,0.6875*length(A)+1.21875),
       cost(ub,michelson_steps,80*length(A)+140) ).
```

Listing 1.14. Analysis output for contract 1.1.

recurrence equations to obtain output argument sizes as functions of input sizes. We use a hierarchical recurrence solver that classifies the equations and dispatches them to an internal solver or interfaces with existing tools like Mathematica, PURRS, PUBS, Matlab, etc., and also combine with techniques such as ranking functions.

3. **Resource analysis**: Use the size information to set up recurrence equations representing the resource consumptions of each version of each predicate (block), and again compute bounds to their solutions, as above, to obtain the output resource usage bound functions.

In the `CiaoPP` implementation all of these analysis tasks are performed by the PLAI abstract interpretation framework [49, 30] of `CiaoPP`, using different *abstract domains* (Fig. 4). The generic resource analysis is also fully based on

abstract interpretation [12] and defined as a PLAI-style abstract domain of piecewise functions and equations [59]. This brings in features such as *multivariance*, efficient fixpoints, assertion-based verification and user interaction, etc.

Michelson Contract Analysis Example. As an example of the analysis process, we analyze the contract of Listing 1.1. In the CHC IR representation of the contract in Listing 1.13, we can observe how the translation has generated a predicate with two clauses that emulates the semantics of the `ITER` instruction: it takes the list over which to iterate as a parameter and performs the `CONS` action specified by the body of the `ITER` instruction. In both clauses the translation tool includes a cost marker to measure the cost of each iteration step, and of leaving the loop. The output from `CiaoPP`, after performing analyses for shapes/measures, sharing, non-failure, sizes, and resources is shown in Listing 1.14. The cost in *gas* of this contract is inferred to be linear w.r.t. the length of the input list.

6 Some experimental results

We have constructed a prototype which transforms Michelson contracts to CHC IR, as well as the cost model that provides `CiaoPP` with the required information on the Michelson instructions. This cost model contains 97 cost assertions, covering a large percentage of Michelson instructions, and is easy to extend, as shown in Section 4.

Regarding the translator, it is 700 lines long, of which 190 correspond to instruction definitions, transliterated from the specification, and 175 to instruction metadata. The whole system was developed in about two months. In our prototype and experiments we have concentrated on the gas cost of *executing* a contract. However, we believe that the framework can be instantiated to other costs such as *type checking* or *storage size*, using the *sized types*-based analyses in the system [58, 59].

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. Results for a selection are listed in Table 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.[8] Column **Contract** lists the contracts, 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 for brevity just the order of the resource usage function inferred by the analysis in terms of the sizes of the parameter (α) and the storage (β) or k if the inferred function is constant. However, the actual expressions inferred also include the constants. For complex metrics, sub-indices 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. Finally, **Time** shows the time

[8] They benchmarks themselves are briefly explained in Table 2 in the Appendix.

Contract	Metrics		Resource A.	Time
	Parameter (α)	Storage (β)	*gas*	(ms)
`reverse`	*length*	*length*	α	216
`addition`	*value*	*value*	$\log\alpha$	147
`michelson_arith`	*value*	*value*	$\log(\alpha^2 + 2 * \beta)$	208
`bytes`	*value*	*length*	β	229
`list_inc`	*value*	*length*	β	273
`lambda`	*value*	*value*	$\log\alpha$	99
`lambda_apply`	$(value, size)$	*size*	k	114
`inline`	*size*	*value*	$\log\beta$	870
`cross_product`	$(length, length)$	*value*	$\alpha_1 + \alpha_2$	424
`lineal`	*value*	*value*	α	244
`assertion_map`	$(value, size)$	*length*	$\log\beta * \log\alpha_1$	393
`quadratic`	*length*	*length*	$\alpha * \beta$	520
`queue`	*size*	$(value, size, length)$	$\log\beta_1 * \log\beta_3$	831
`king_of_tez`	*size*	$(value, value, size)$	k	635
`set_management`	*length*	*length*	$\alpha * \log\beta$	357
`lock`	*size*	$(value, value, size)$	k	421
`max_list`	*length*	*size*	α	473
`zipper`	*length*	$(length, length, length)$	k	989
`auction`	*size*	$(value, value, size)$	k	573
`union`	$(length, length)$	*length*	$\alpha_1 * \log\alpha_2$	486
`append`	$(length, length)$	*length*	α_1	371
`subset`	$(length, length)$	*size*	$\alpha_1 * \log\alpha_2$	389

Table 1. Results of analysis for selected Michelson contracts.

taken to perform all the analyses using the different abstract domains provided by `CiaoPP`, version 1.19 on a medium-loaded 2.3 GHz Dual-Core Intel Core i5, 16 GB of memory, running macOS Catalina 10.15.6. 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 the approach.

7 Related Work

As mentioned in the introduction, the tools that have been proposed to date for resource analysis of smart contracts are platform- and language-specific. GASPER [10] and MadMax [22] are both aimed at identifying parts of contracts that have high gas consumption in order to optimize them or to avoid gas-related vulnerabilities. GASPER is based on recognizing control-flow patterns using symbolic computation while MadMax searches for both control- and

data-flow patterns. Marescotti et al. [44] also use a limited-depth path exploration approach to estimate worst-case gas consumption. These tools are useful programmer aids for finding bugs, but cannot provide safe cost bounds. GASPER and MadMax are specific to contracts written for the Ethereum platform [63], in Solidity, and translated to Ethereum Virtual Machine (EVM) bytecode. The Solidity compiler can generate gas bounds, but these bounds can only be constant, i.e., they cannot depend on any input parameters, or if they do the bound generated is infinite. This tool is of course also specific to the Ethereum platform.

Closer to our work are GASTAP [5] and its extension GASOL [4]. These tools infer upper bounds for gas consumption, using similar theoretical underpinnings 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 optimization 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 Resource Analysis (also referred to as user-defined resource analysis) was proposed in [52] and developed further in [51, 59]. The approach builds on Wegbreit's seminal work [62] and the first full analyzers for upper bounds, in the context of task granularity control in automatic program parallelization [15, 14]. This in turn evolved to cover other types of approximations (e.g., lower bounds [16]), and to the idea of supporting resources defined at the user level [52, 51]. This analysis was extended to be fully based on abstract interpretation [12] and integrated into the PLAI multi-variant framework, leading to context-sensitive resource analyses [59]. Other extensions include static profiling [43], static bounding of run-time checking overhead [38], or analysis of parallel programs [37]. Other applications include the previously mentioned analyses of platform-dependent properties such as time or energy [50, 47, 51, 41, 40, 39, 42].

Resource analysis has received considerable additional attention lately [61, 31, 23, 33, 53, 19, 3, 24, 31, 45, 32, 11, 7, 8, 57, 36, 48, 26]. 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.

8 Conclusions and Future Work

We have explored the application of a generic approach and tool for resource consumption analysis to the problem of static inference of gas consumption bounds in smart contracts. The objective has been to provide a quick development path for cost analyses for new smart contract platforms and languages, or easily adapting existing ones to changes. To this end, we have used the techniques of Parametric Resource Analysis and translation to Horn clause-based intermediate representations, using the `Ciao`/`CiaoPP` system as tool and the Tezos platform and its Michelson language as test cases. The Horn clause translator together with the cost model and `Ciao`/`CiaoPP` constitute a gas consump-

tion analyzer for Tezos smart contracts. We also applied this tool to a series of smart contracts obtaining relevant bounds with reasonable processing times. We believe our experience and results are supportive of our hypothesis that this general approach allows 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. In fact, while preparing the final version of this paper, a new protocol, *Delphi*, was released and we were able to update the cost model in less than a day by modifying just the cost assertions. As a final remark, we would also like to point out that the approach and tools that we have used bring in much additional functionality beyond that discussed herein, which is inherited from the `Ciao`/`CiaoPP` framework used, such as resource usage certification, static debugging of resource consumption, static profiling, or abstraction-carrying code.

References

1. The Michelson Language Site, https://www.michelson-lang.com
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning **46**(2), 161–203 (2011)
3. Albert, E., Genaim, S., Masud, A.N.: More Precise yet Widely Applicable Cost Analysis. In: Proc. of VMCAI'11. LNCS, vol. 6538, pp. 38–53. Springer (2011)
4. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: GASOL: gas analysis and optimization for ethereum smart contracts. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020. LNCS, vol. 12079, pp. 118–125. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_7
5. Albert, E., Gordillo, P., Rubio, A., Sergey, I.: Running on fumes - preventing out-of-gas vulnerabilities in ethereum smart contracts using static resource analysis. In: VECoS 2019. LNCS, vol. 11847, pp. 63–78. Springer (October 2019). https://doi.org/10.1007/978-3-030-35092-5_5
6. Allombert, V., Bourgoin, M., Tesson, J.: Introduction to the tezos blockchain. CoRR **abs/1909.08458** (2019), http://arxiv.org/abs/1909.08458
7. Avanzini, M., Lago, U.D.: Automating sized-type inference for complexity analysis. Proc. ACM Program. Lang. **1**(ICFP), 43:1–43:29 (2017). https://doi.org/10.1145/3110287
8. Blazy, S., Pichardie, D., Trieu, A.: Verifying constant-time implementations by abstract interpretation. In: European Symposium on Research in Computer Security – ESORICS 2017. Lecture Notes in Computer Science, vol. 10492, pp. 260–277. Springer (September 2017). https://doi.org/10.1007/978-3-319-66402-6_16
9. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M.V., Lopez-Garcia, P., Puebla-(Eds.), G.: The Ciao System. Ref. Manual (v1.13). Tech. rep., School of Computer Science, T.U. of Madrid (UPM) (2009), available at http://ciao-lang.org
10. Chen, T., Li, X., Luo, X., Zhang, X.: 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). https://doi.org/10.1109/SANER.2017.7884650
11. Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational cost analysis. In: Castagna, G., Gordon, A.D. (eds.) Principles of Programming Languages,

POPL 2017. pp. 316–329. ACM (2017), http://dl.acm.org/citation.cfm?id=3009858
12. Cousot, P., Cousot, R.: 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)
13. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Semantics-based generation of verification conditions by program specialization. In: 17th International Symposium on Principles and Practice of Declarative Programming. pp. 91–102. ACM (July 2015). https://doi.org/10.1145/2790449.2790529
14. Debray, S.K., Lin, N.W.: Cost Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems **15**(5), 826–875 (November 1993)
15. Debray, S.K., Lin, N.W., Hermenegildo, M.V.: 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)
16. Debray, S.K., Lopez-Garcia, P., Hermenegildo, M.V., Lin, N.W.: Lower Bound Cost Estimation for Logic Programs. In: 1997 International Logic Programming Symposium. pp. 291–305. MIT Press, Cambridge, MA (October 1997)
17. Futamura, Y.: Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. Systems, Computers, Controls **2**(5), 45–50 (1971)
18. Gallagher, J., Hermenegildo, M.V., Kafle, B., Klemen, M., Lopez-Garcia, P., Morales, J.: From big-step to small-step semantics and back with interpreter specialization (invited paper). In: International WS on Verification and Program Transformation (VPT 2020). pp. 50–65. EPTCS, Open Publishing Association (2020), http://eptcs.web.cse.unsw.edu.au/paper.cgi?VPTHCVS2020.4
19. Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F., Fuhs, C.: Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In: Proceedings of PPDP'12. pp. 1–12. ACM (2012)
20. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Decompilation of Java Bytecode to Prolog by Partial Evaluation. JIST **51**, 1409–1427 (October 2009)
21. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12. pp. 405–416. ACM (2012). https://doi.org/10.1145/2254064.2254112
22. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: MadMax: surviving out-of-gas conditions in ethereum smart contracts. PACMPL **2**(OOPSLA), 116:1–116:27 (2018). https://doi.org/10.1145/3276486
23. Grobauer, B.: Cost recurrences for DML programs. In: Proceedings of ICFP '01. pp. 253–264. ACM, New York, NY, USA (2001). https://doi.org/10.1145/507635.507666, http://doi.acm.org/10.1145/507635.507666
24. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: 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)
25. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: International Conference on Computer Aided Verification, CAV 2015. pp. 343–361. No. 9206 in LNCS, Springer (July 2015)
26. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate your assets: reasoning about resource usage in liquid haskell. Proc. ACM Program. Lang. **4**(POPL), 24:1–24:27 (2020). https://doi.org/10.1145/3371092
27. Henriksen, K.S., Gallagher, J.P.: Abstract Interpretation of PIC Programs through Logic Programming. In: SCAM '06. pp. 184–196. IEEE Computer Society (2006)

28. Hermenegildo, M.V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. TPLP **12**(1–2), 219–252 (2012), http://arxiv.org/abs/1102.5497
29. Hermenegildo, M.V., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2), 115–140 (October 2005). https://doi.org/10.1016/j.scico.2005.02.006
30. Hermenegildo, M.V., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM TOPLAS **22**(2), 187–223 (March 2000)
31. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM TOPLAS **34**(3), 14:1–14:62 (2012)
32. Hofmann, M., Moser, G.: Multivariate amortised resource analysis for term rewrite systems. In: Altenkirch, T. (ed.) 13th International Conference on Typed Lambda Calculi and Applications. LIPIcs, vol. 38, pp. 241–256. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (July 2015). https://doi.org/10.4230/LIPIcs.TLCA.2015.241
33. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Symposium on Principles of Programming Languages. pp. 331–342. ACM (2002), citeseer.ist.psu.edu/igarashi02resource.html
34. Kahn, G.: Natural semantics. Lecture Notes in Computer Science, vol. 247, pp. 22–39. Springer (February 1987). https://doi.org/10.1007/BFb0039592, https://doi.org/10.1007/BFb0039592
35. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016. LNCS, vol. 9779, pp. 352–358. Springer (July 2016). https://doi.org/10.1007/978-3-319-41528-4_19
36. Kincaid, Z., Breck, J., Cyphert, J., Reps, T.W.: Closed forms for numerical loops. Proc. ACM Program. Lang. **3**(POPL), 55:1–55:29 (2019). https://doi.org/10.1145/3290368
37. Klemen, M., Lopez-Garcia, P., Gallagher, J., Morales, J., Hermenegildo, M.V.: A General Framework for Static Cost Analysis of Parallel Logic Programs. In: International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19). LNCS, vol. 12042, pp. 19–35. Springer-Verlag (April 2020). https://doi.org/10.1007/978-3-030-45260-5_2
38. Klemen, M., Stulova, N., Lopez-Garcia, P., Morales, J.F., Hermenegildo, M.V.: 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). https://doi.org/10.1145/3236950.3236970
39. Liqat, U., Banković, Z., Lopez-Garcia, P., Hermenegildo, M.V.: Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks. In: Logic-Based Program Synthesis and Transformation - 27th International Symposium. LNCS, vol. 10855. Springer (2018)
40. Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M.V., Gallagher, J.P., Eder, K.: Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In: Proc. of FOPARA. LNCS, vol. 9964, pp. 81–100. Springer (2016). https://doi.org/10.1007/978-3-319-46559-3_5
41. Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M.V., Eder, K.: Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In: Proceedings of LOPSTR'13. LNCS, vol. 8901, pp. 72–90. Springer (2014). https://doi.org/10.1007/978-3-319-14125-1_5

42. Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., Hermenegildo, M.V.: 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 **18**(2), 167–223 (March 2018), https://arxiv.org/abs/1803.04451
43. Lopez-Garcia, P., Klemen, M., Liqat, U., Hermenegildo, M.V.: A General Framework for Static Profiling of Parametric Resource Usage. TPLP (ICLP'16 Special Issue) **16**(5-6), 849–865 (2016). https://doi.org/10.1017/S1471068416000442
44. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018). LNCS, vol. 11247, pp. 450–465. Springer (November 2018). https://doi.org/10.1007/978-3-030-03427-6_33
45. Maroneze, A.O., Blazy, S., Pichardie, D., Puaut, I.: A formally verified WCET estimation tool. In: Workshop on Worst-Case Execution Time Analysis – WCET 2014. OASICS, vol. 39, pp. 11–20. Schloss Dagstuhl (2014). https://doi.org/10.4230/OASIcs.WCET.2014.11
46. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: LOPSTR. LNCS, vol. 4915, pp. 154–168. Springer-Verlag (August 2007). https://doi.org/10.1007/978-3-540-78769-3_11
47. Mera, E., Lopez-Garcia, P., Carro, M., Hermenegildo, M.V.: Towards Execution Time Estimation in Abstract Machine-Based Languages. In: PPDP'08. pp. 174–184. ACM Press (July 2008). https://doi.org/10.1145/1389449.1389471
48. Moser, G., Schneckenreither, M.: Automated amortised resource analysis for term rewrite systems. Sci. Comput. Program. **185** (2020). https://doi.org/10.1016/j.scico.2019.102306
49. Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. Journal of Logic Programming **13**(2/3), 315–347 (July 1992)
50. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: 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
51. Navas, J., Méndez-Lojo, M., Hermenegildo, M.V.: User-Definable Resource Usage Bounds Analysis for Java Bytecode. In: BYTECODE'09. ENTCS, vol. 253, pp. 6–86. Elsevier (March 2009), http:cliplab.org/papers/resources-bytecode09.pdf
52. Navas, J., Mera, E., Lopez-Garcia, P., Hermenegildo, M.: User-Definable Resource Bounds Analysis for Logic Programs. In: Proc. of ICLP'07. LNCS, vol. 4670, pp. 348–363. Springer (2007). https://doi.org/10.1007/978-3-540-74610-2_24
53. Nielson, F., Nielson, H., Seidl, H.: Automatic complexity analysis. In: Programming Languages and Systems, pp. 243–261. LNCS, Springer (2002)
54. Peralta, J., Gallagher, J., Sağlam, H.: Analysis of imperative programs through analysis of constraint logic programs. In: Levi, G. (ed.) Static Analysis. 5th International Symposium, SAS'98, Pisa. LNCS, vol. 1503, pp. 246–261 (1998)
55. Plotkin, G.: A structural approach to operational semantics. Technical report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark (1981)
56. Puebla, G., Bueno, F., Hermenegildo, M.V.: An Assertion Language for Constraint Logic Programs. In: Analysis and Visualization Tools for Constraint Programming, pp. 23–61. No. 1870 in LNCS, Springer-Verlag (2000)

57. Qu, W., Gaboardi, M., Garg, D.: Relational cost analysis for functional-imperative programs. Proc. ACM Program. Lang. **3**(ICFP), 92:1–92:29 (2019). https://doi.org/10.1145/3341696
58. Serrano, A., Lopez-Garcia, P., Bueno, F., Hermenegildo, M.V.: Sized Type Analysis for Logic Programs (technical communication). In: Swift, T., Lamma, E. (eds.) Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement. vol. 13, pp. 1–14. Cambridge U. Press (August 2013)
59. Serrano, A., Lopez-Garcia, P., Hermenegildo, M.V.: Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. TPLP, ICLP'14 Special Issue **14**(4-5), 739–754 (2014). https://doi.org/10.1017/S147106841400057X
60. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997). https://doi.org/10.5210/fm.v2i9.548
61. Vasconcelos, P., Hammond, K.: Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In: IFL'03. LNCS, vol. 3145, pp. 86–101. Springer (2003)
62. Wegbreit, B.: Mechanical Program Analysis. Communications of the ACM **18**(9), 528–539 (September 1975)
63. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2016), https://gavwood.com/paper.pdf

A Brief description of selected Michelson contracts

Contract	Overview
`reverse`	Reverses the input list and stores the result.
`addition`	Performs a simple Michelson addition.
`michelson_arith`	Calculates the function: $f(x, y) = x^2 + 2 * y + 1$.
`bytes`	Slices the bytes storage according to the provided parameter.
`list_inc`	Increments list of numbers in the storage by the provided parameter.
`lambda`	Runs a lambda function passing the parameter as argument.
`lambda_apply`	Specializes the provided lambda function and creates a Michelson operation.
`inline`	Runs a lambda function several times passing different arguments.
`cross_product`	Performs the cross product of the lists passed as parameters.
`linear`	Loops over a number.
`assertion_map`	Performs a series of operations on a Michelson map.
`quadratic`	Loops over the parameter and storage lists.
`queue`	Implements a queue in which calls can push or pop elements.
`king_of_tez`	Stores the identity of the highest bidder.
`set_management`	Iterates the input list from left to right and removes from the storage set those elements already in it and inserts those which are not present yet.
`lock`	Implements a lock on a contract.
`max_list`	Obtains the largest number in a list.
`zipper`	Implements a zipper data structure.
`auction`	Implements a distributed auction with a time limit.
`union`	Calculates the union of two sets.
`append`	Appends two input lists.
`subset`	States whether an input set is a subset of the other.

Table 2. Overview of the selected Michelson contracts.