特別研究報告書

スマートコントラクトのガス消費量の Resource Aware MLを用いた静的解析

指導教員：末永　幸平　准教授

京都大学工学部情報学科

小野　雄登

2021年2月2日

スマートコントラクトのガス消費量の Resource Aware ML を用いた静的解析

小野　雄登

内容梗概

2008 年にビットコインが開発されて以来，現在に至るまでにブロックチェーンを技術基盤とする様々な仮想通貨が開発されている．スマートコントラクトは，仮想通貨の取引における契約の締結や履行を自動化する仕組みであり，ブロックチェーン上で動作するプログラムとして実装される．

スマートコントラクトには**ガス**の概念が存在する．ガスはコントラクトの実行のために利用する計算資源にかかる手数料を表している．コントラクトが実行される際に，コントラクトの各命令の実行毎に命令の計算コストに応じた量のガスが消費される．消費量の合計が許容ガス消費量を超えると，プログラムの実行が直ちに停止され，プログラムの実行による効果がロールバックされる．コントラクトを実行しようとする際は，あらかじめ一定量のガスに相当する通貨を支払う必要があるが，これは実行が取り消されても返金されないため，実行コストが無駄にかかってしまうことになる．コントラクトの実行コストを抑えるために，ガスの消費量を静的に解析する手法が求められている．

プログラムの実行コストを解析する手段の一つとして，Resource Aware ML(RAML) がある．RAML は，ポテンシャルベースの償却解析と呼ばれる，データ構造の状態に応じてコストが変わる一連の操作のコストを解析する解析技術をもとに設計された，OCaml で用いられる文法を備えた関数型プログラミング言語である．RAML は入力として与えられたプログラムのリソース消費量の上界を，指定されたメトリックに従って自動的に，かつ静的に解析して，その結果を出力するツールとして用いることができる．

本研究では，RAML を用いて，仮想通貨 Tezos のスマートコントラクトのガス消費量を静的に解析する手法を提案する．Tezos のスマートコントラクトは，スタックベースのプログラミング言語 Michelson で書かれている．そのために，Michelson において実装されている各命令の挙動を模倣するライブラリを RAML において設計する．このライブラリを用いて，Michelson プログラムの挙動を模倣する RAML プログラムを作成することができる．また，そのプログラムを RAML で解析することで，コントラクトのガス消費量を見積もること

ができる．

ライブラリにおいては，Michelson の命令のうち主要なものについて，命令を模倣する関数を RAML で設計した．Michelson の命令は，スタックを受け取ってスタックの内容を書き換える．これを RAML で設計するにあたって，スタックの要素をヴァリアント型 `t` として定義し，スタックを型 `t` のリストとして扱い，命令を (`t list -> t list`) 型をもつ関数として定義した．Michelson のコントラクトは，初期スタックに積まれる値の型宣言と，初期スタックに対して順に適用される一連の命令によって構成されている．Michelson のコントラクトの動作を模倣するプログラムを，本ライブラリを用いて，初期スタックを表すリストに対して命令を順に関数適用するプログラムとして実装することができる．

コントラクトのガス消費量の見積もりについては，Tezos のコントラクトの実行において発生するガスの消費量はいくつかのコストに分けられているが，そのうち，プログラムの解釈実行を行う際に発生する Interpreter Cost を見積もることを目的に取り組んだ．方法としては，RAML の tick メトリックを用いる．tick メトリックは，リソース消費の値や発生するタイミングを，ユーザーが関数として定義することができるメトリックである．本ライブラリの各命令について，その命令の Interpreter Cost に相当する値の tick 関数を定義し，tick メトリックを用いて，コントラクトを模倣するプログラムの解析を行い，コントラクトの Interpreter Cost を見積もる．

いくつかの Michelson コントラクトを模倣する RAML プログラムを作成し，それに対して解析を行いガス消費量の見積もりを行った．その結果，基本的なスタック操作や条件分岐の命令のみで構成されているコントラクトについては解析が成功し，Interpreter Cost を正しく見積もることができた．一方で，リストや集合に対する再帰を行う命令を含むコントラクトでは解析が行えなかった．また，スタックの内容によってガス消費量が異なるような命令を含むコントラクトについては，正しく Interpreter Cost の見積もりを行うことは難しかった．

本研究において RAML で実装しなかった命令の実装や，解析が正しく行えなかった命令の再実装については，今後の課題とする．また，コントラクトの実行において発生するその他のコストの見積もり，ひいてはコントラクトの実行コスト全体の見積もりについても検討していきたい．

Static Analysis for Gas Consumption of Smart Contracts Using Resource Aware ML

Yuto Ono

Abstract

Since the development of BitCoin in 2008, a variety of cryptocurrencies based on blockchain technology have been developed. A smart contract is a mechanism to automate the conclusion and fulfillment of contracts in cryptocurrency transactions and is implemented as a program that executes on a blockchain.

Smart contracts have a concept of *gas*. The gas represents the fee for the computational resources used to execute the contract. When a contract is executed, the execution of each instruction consumes an amount of gas corresponding to the computational cost of the instruction. If the total consumption exceeds its allowed gas consumption, the execution of the program is stopped immediately and the effect of the program execution is rolled back. When executing a contract, the executor needs to pay the currency equivalent to a certain amount of gas in advance, which is not refunded even if the execution is stopped, resulting in unnecessary execution costs. A method to statically analyze the gas consumption is required to reduce the execution cost of the contract.

Resource Aware ML (RAML) is one of the methods to analyze the execution cost of programs. RAML is a functional programming language with OCaml grammar, based on the analysis technique called potential-based amortized analysis which analyzes the costs of a sequence of operations whose costs vary depending on the state of the data structure, RAML can be used as a tool to automatically and statically analyze the resource consumption bounds for programs given as input according to a specific metric and output the result.

This research proposes a method to statically analyze the gas consumption of smart contracts on cryptocurrency Tezos using RAML. Tezos smart contract is written in Michelson, a stack based programming language. To do this, we design a library in RAML that imitates the behavior of each instruction

implemented in Michelson. Using this library, we can write RAML programs that imitate the behavior of Michelson programs. Then we can estimate the gas consumption of the contract by analyzing this program with RAML.

In the library, we design functions that imitate core Michelson instructions in RAML. The Michelson instruction takes a stack and rewrites the content of it. To design it in RAML, we defined the elements of a stack as variant type `t`, treated a stack as a list of type `t`, and defied an instruction as a function of type `(t list -> t list)`. A Michelson contract consists of a type declaration of the values piled on the initial stack and a sequence of instructions to be applied to the initial stack. we can implement a program that imitates the behavior of a Michelson contract as a program that applies instructions in sequence to a list that represents the initial stack, using this library.

As for the estimation of the gas consumption of the contract, the gas consumption of execution of the Tezos contract is divided into several costs, and we aimed to estimate the Interpreter Cost that occurs in the interpretation and execution of the program. We use the tick metric of RAML, which allows the user to define the value of resource consumption and the timing of its occurrence as a function. For each instruction in the library, we define a tick function with a value corresponding to its Interpreter Cost and use the tick metric to analyze a program that imitates the contract to estimate the Interpreter Cost of the contract.

We implemented several RAML programs that imitate Michelson contracts and analyzed them to estimate the gas consumption. As a result, the analysis was successful for the contracts that consisted of only instructions of basic stack operations and conditional branch, and we could estimate the Interpreter Cost correctly. On the other hand, the analysis was not successful for contracts that contain instructions for recursion on lists and sets. Also, it was difficult to correctly estimate the Interpreter Cost of the contracts that contain instructions that consume gas depending on the contents of the stack.

The implementation of instructions that were not implemented in this paper and the reimplementation of instructions that could not be analyzed correctly are future tasks. And we would like to consider the estimation

of other costs incurred in the execution of a contract, and eventually the estimation of the overall execution cost of a contract.

スマートコントラクトのガス消費量の Resource Aware ML を用いた静的解析

目次

1 序論

2009年にビットコインを用いた取引がオープンソフトウェアで始まって以来，現在に至るまでにブロックチェーンを技術基盤とする様々な仮想通貨が開発されている．取引の記録をブロックとしてネットワーク上に記憶するという性質上，ブロックチェーンはデータ改竄に対する優れた耐性を持ち，仮想通貨の取引を支えるコア技術となっている．

ブロックチェーン上で用いられる技術としてスマートコントラクトがある．スマートコントラクトは，仮想通貨の取引における契約の締結や履行を自動化する仕組みであり，ブロックチェーン上で動作するプログラムとして扱われる．第3者を介さずに，また相手の信頼を必要とせずに取引を行うことができ，決済期間の短縮や手数料の削減などの効果が期待できる．スマートコントラクトにはガスの概念が存在し，ガスはコントラクトの実行にかかる手数料を表している．コントラクトが実行される際に，コントラクトの各命令の評価毎に命令の内容に比例した量のガスが消費され，消費量の合計が許容ガス消費量を超えると，その命令が直ちに停止され，命令の実行による値の変更が取り消される．ガスの消費量の計算は複雑で，前もってガスの消費量を正確に見積もることは難しいとされているが，プログラムとして非効率なコントラクトが実行されると，想定される量以上のガスが消費されてしまうので，コントラクトのガス消費量を静的に解析することは，ユーザーが必要以上に手数料を支払わないために必要な技術であると考えられる．

本研究では，仮想通貨Tezosのスマートコントラクトのガス消費量の静的な解析を行うプログラムを実装した．Tezosはスマートコントラクトを用いたブロックチェーンを技術基盤とする仮想通貨の1つで，コントラクトはスタックベースのプログラミング言語Michelsonで書かれている．コントラクトのガス消費量はMichelsonプログラムの実行内容によって計算されるので，このプログラムに対して解析を行うことでガス消費量の見積もりを試みた．

解析の方法として，プログラミング言語型のツールであるResource Aware ML(RAML)を用いる．RAMLは，OCamlで用いられる文法を備えた関数型プログラミング言語で，入力として与えられたプログラムのリソース消費量の上界を，指定されたメトリックに従って自動的に，かつ静的に解析して，その結果を出力するツールである．具体的な方法としては，Michelsonで実装されて

いる型や命令などのライブラリを RAML で実装し，Michelson のコントラクトを RAML 上でエンコードし，それを解析してガス消費量の見積もる．

本報告書は以下のように構成されている．第 2 節では，本研究の背景知識として，Tezos と Michelson，スマートコントラクトにおけるガス消費の仕組み，そして解析に用いるツールである RAML についてそれぞれ説明する．第 3 節では，Mihcelson プログラムの RAML での実装について説明する．第 4 節では，第 3 章で実装した RAML プログラムを用いて行ったガス消費量の解析について，結果と考察を記述する．第 5 節では，実装したプログラムについていくつかの改善点を示す．最後に第 6 節で本研究についての結論を述べる．

2　背景知識

本節では，本研究に関連する背景知識について述べる．第 3.1 節では Tezos と Michelson について，第 3.2 節ではスマートコントラクトにおけるガス消費の仕組みについて，第 3.3 節では RAML についてそれぞれ述べる．

2.1　Tezos と Michelson について

Tezos はスマートコントラクトを用いたブロックチェーンを技術基盤とする仮想通貨の 1 つである．Bitcoin や Ethereum といった仮想通貨が先立って流通されるようになった中，Tezos はそれらのブロックチェーンの弱点を解消することを目的として開発された．Tezos の特徴として"Proof-of-Stake"(PoS) と呼ばれるコンセンサスアルゴリズムを採用していることが挙げられる．従来のブロックチェーンで採用されている"Proof-of-Work"(PoW) が，コンピュータの計算能力が高いユーザーに対してブロック生成の権利を与えているのに対して，PoS では通貨の保有量が多いユーザーに対してブロック生成の権利が与えられる．Tezos の採用している PoS は"Liquid-Proof-of-Stake"(LPoS) といい，ブロック生成の権利を他のユーザーに委任することができる．これにより，多くのユーザーがブロック生成に参加することができ，プロトコルの分散性を高めるという目的に寄与している．

Mihcelson は，Tezos のスマートコントラクトを記述するために用いられるプログラミング言語である．この言語はスタックベースで，高レベルのデータ型とプリミティブ，および厳密な静的型チェックを備えている．

Michelson の文法のうち，本研究で RAML で実装する部分の文法を図**??**に示す．T はスタックの要素の型を表す．T は整数値の型 int，自然整数の型 nat，Tezos における通貨量の型 mutez，Boolean の型 bool，アドレスを表す型 address，Unit 値の型 unit，オプション値の型 option，命令を表す型 operation，コントラクトを表す型 contract，ペアの型 pair，ユニオンの型 or がある．V はスタックの要素になり得る値を表し，整数値の i，Boolean の True，False，アドレスの a，Unit 値の Unit，ペアの (V_1, V_2)，ユニオンの Left V，Right V，オプション値の Some V，None，空リストの [ ]，リストの結合を表す $V_1 :: V_2$，命令列 IS がある．n は自然整数，i は整数である．IS は命令列で，命令 I のシーケンスである．命令 I は，プログラムを中止する命令 (FAILWITH)，構造のコントロールに関する命令 (IF，LOOP，DIP)，スタックの操作に関する命令 (DROP，DUP，SWAP，PUSH，UNIT)，スタックのトップ要素の比較に関する命令 (EQ，NEQ，LT，GT，LE，GE)，Boolean に関する操作の命令 (OR，AND，XOR，NOT)，整数値に関する命令 (NEG，ABS，ISNAT，INT，ADD，SUB，MUL，EDIV)，スタックの上から 2 つの要素の比較命令 (COMPARE)，ペアに関する命令 (PAIR， CAR，CDR)，オプション値に関する命令 (SOME，NONE，IF_NONE)，ユニオンに関する命令 (LEFT，RIGHT，IF_LEFT)，リストに関する命令 (CONS，NIL，IF_CONS，MAP，SIZE，ITER)，コントラクトに関する命令 (CONTRACT，TRANSFER_TOKENS，AMOUNT，SOURCE) がある．S はスタックを表す．空のスタックを E で表し，スタックに積まれた要素は : で区切る．

$$
\begin{aligned}
T &::= \text{int} \mid \text{nat} \mid \text{mutez} \mid \text{bool} \mid \text{address} \mid \text{unit} \mid \\
&\quad \text{option } T \mid \text{list } T \mid \text{operation} \mid \text{contract } T \mid \text{pair } T\ T \mid \text{or } T\ T \\
V &::= i \mid \text{True} \mid \text{False} \mid a \mid \text{Unit} \mid (V_1, V_2) \mid \text{Left } V \mid \text{Right } V \mid \\
&\quad \text{Some } V \mid \text{None} \mid [\,] \mid V_1 :: V_2 \mid IS \\
n &::= [0-9]+ \\
i &::= n \mid -n \\
IS &::= \{I_1;\ \ldots ; I_n\} \\
I &::= \text{FAILWITH} \mid \text{IF } IS_1\ IS_2 \mid \text{LOOP } IS \mid \text{DIP } IS \mid \\
&\quad \text{DROP} \mid \text{DUP} \mid \text{SWAP} \mid \text{PUSH } T\ V \mid \text{UNIT} \mid \\
&\quad \text{EQ} \mid \text{NEQ} \mid \text{LT} \mid \text{GT} \mid \text{LE} \mid \text{GE} \mid \text{OR} \mid \text{AND} \mid \text{XOR} \mid \text{NOT} \mid \\
&\quad \text{NEG} \mid \text{ABS} \mid \text{ISNAT} \mid \text{INT} \mid \text{ADD} \mid \text{SUB} \mid \text{MUL} \mid \text{EDIV} \mid \\
&\quad \text{COMPARE} \mid \text{PAIR} \mid \text{CAR} \mid \text{CDR} \mid \text{SOME} \mid \text{NONE } T \mid \\
&\quad \text{IF\_NONE } IS_1\ IS_2 \mid \text{LEFT } T \mid \text{RIGHT } T \mid \text{IF\_LEFT } IS_1\ IS_2 \mid \\
&\quad \text{NIL } T \mid \text{CONS} \mid \text{IF\_CONS } IS_1\ IS_2 \mid \text{SIZE} \mid \text{MAP } IS \mid \text{ITER } IS \mid \\
&\quad \text{CONTRACT } T \mid \text{TRANSFER\_TOKENS} \mid \text{AMOUNT} \mid \text{SOURCE} \\
S &::= \text{E} \mid V : S
\end{aligned}
$$

Michelsonのプログラムは，プログラムに対して与えられるパラメータと，ブロックチェーン上に保存されているストレージのペアを受け取り，このプログラムの終了後に実行される操作のリストと，プログラムの実行中に更新されブロックチェーン上に保存されるストレージのペアを返す純粋な関数である．プログラムの本体は，順番に実行される一連の命令である．各命令は，スタックを入力として受け取り，そのスタックの内容を書き換えて出力する関数である．プログラムの初期スタックは，引数として与えられたパラメータとストレージのペアが一番上に積まれた状態のもので，その初期スタックに対して順番に命令が適用され，最後に操作のリストとストレージのペアが一番上に積まれた状態のスタックが残り，それが出力される．

Michelsonプログラムの例として，簡単な演算を行うプログラムであるexample1のコードをCode 1に示す．これは，整数のペアをパラメータとして受

け取り，2つの整数の和をストレージに書き込むプログラムである．なお，各命令後のスタックの状態を/*,*/で囲われたコメントで示している．

```
1  parameter (pair int int);
2  storage int;
3  code  /* ((para1, para2), st) */
4    { CAR ;   /* (para1, para2) */
5      DUP ;   /* (para1, para2) : (para1, para2) */
6      CAR ;   /* para1 : (para1, para2) */
7      DIP { CDR } ;   /* para1 : para2 */
8      ADD ;   /* st' */
9      NIL operation ;   /* [] : st' */
10     PAIR    /* ([], st') */
11   }
```

Code 1: example1.tz

以下，プログラムの内容を説明する．

Michelson プログラムのコードは，プログラムに対して与えられる引数である parameter と，ブロックチェーン上に保存されているストレージの値である storage の型宣言から始まる．1行目は parameter の型が (pair int int) であること，2行目は storage の型が int であることを宣言している．3行目以降はプログラムの本体である code である．code には一連の命令が記述されていて，この命令が初期スタックに対して順に実行されていく．以下，code の内容について行番号ごとに説明する．

- プログラム開始時のスタックは，parameter と storage の値のペアが空のスタックに積まれた状態で始まる．parameter の値を (para1, para2)，storage の値を st と表すとすると，初期スタックは ((para1, para2), st) である．
- 4行目の CAR は，スタックのトップの要素がペアだった場合，その要素を取り出して，ペアの第1要素をスタックに積む命令である．4行目時点でのスタックは (para1, para2) である．
- 5行目の DUP は，スタックのトップの要素を複製してスタックに積む命令である．5行目時点でのスタックは (para1, para2) : (para1, para2) である．
- 6行目は4行目と同じく CAR を実行する．6行目時点でのスタックは para1 : (para1, para2) である．

- 7行目のDIPは引数として命令列bodyを受け取る命令で，スタックのトップの要素を保持した状態で，その要素を取り出した状態のスタックに対してbodyを実行する命令である．つまり，スタックのトップのpara1を保持した状態で，スタック(para1, para2)に対してCDRを実行する．CDRは，スタックのトップの要素がペアの場合，その要素を取り出して，ペアの第2要素をスタックに積む命令である．よって，7行目時点でのスタックはpara1 : para2である．
- 8行目のADDは，スタックのトップの要素xと2番目の要素yの型が，それら2つの演算が定義されているような型である場合，xとyを取り出し，x+yの値をスタックに積む命令である．例えば，xとyがともにint型ならば，x+yはint型である．para1 + para2の値をst′と表すとすると，8行目時点でのスタックはst′である．
- 9行目のNILは引数として型aを受け取る命令で，リストの型がaであるような空のリスト[ ]をスタックに積む命令である．9行目時点でのスタックは[ ] : st′である．
- 10行目のPAIRはスタックのトップの要素と2番目の要素を取り出し，それらのペアをスタックに積む命令である．この命令後，スタックは([ ], st′)となり，プログラムが終了する．

プログラムの終了時のスタックは，このプログラム終了後に続いて行われる操作のリストoperation listと，実行中に更新されブロックチェーンに保存されるストレージの値storage′のペアのみが積まれている状態でなければならない．このとき，プログラム実行前のストレージの値storageと，プログラム実行後のストレージの値storage′の型が一致している必要がある．

Michelsonには静的型チェックが備えられている．それぞれの命令には，命令の実行前と実行後のスタックの状態を型で表した型規則が存在する．例えば，PAIRについての型規則は以下のように表される．

$$\mathrm{a}' : \mathrm{b}' : \mathrm{S}' \to \mathrm{pair}\ \mathrm{a}'\ \mathrm{b}' : \mathrm{S}'$$

これは，実行前のスタックのトップの要素の型がa′，2番目の要素の型がb′のとき，実行後のスタックのトップの要素がpair a′ b′となることを示している．また，ADDについての型規則は以下のように表される．

$$\mathrm{int} : \mathrm{int} : \mathrm{S}' \to \mathrm{int} : \mathrm{S}'$$

これは，実行前のスタックのトップの要素と 2 番目の要素がともに int である必要があり，その場合，実行後のスタックのトップの要素が int となることを示している．

命令を実行する前にスタックの状態が型規則に則った形でない場合，命令は失敗する．これを防ぐために，プログラム実行前に静的な型チェックが行われる．

2.2 コントラクトのガス消費の仕組み

スマートコントラクトにおけるガスは，コントラクトを実行させる上で必要となる手数料を表している．スマートコントラクトを実行する際，ブロックの生成者であるマイナーがそのコントラクトの検証を行い，その対価としてコントラクトの実行者がマイナーに対して手数料を支払う必要がある．この手数料を計算する際にガスという概念が用いられており，計算されたガスの消費量はそのブロックチェーンで用いられる通貨に変換される．

ガス消費量の計算については，コントラクトが実行される際に，コントラクトの各命令の実行毎に命令の計算コストに応じた量のガスが消費される．ガスの消費量の合計が計算されると，それが通貨に変換され，マイナーに対して支払う手数料となる．また，これとは別にあらかじめ一定量のガスに相当する通貨をマイナーに対して支払う必要がある．コントラクトの実行者は実行時にガスの上限値を設定し，ガスの消費量の合計がその上限値を超えると，プログラムの実行が直ちに停止され，プログラムの実行による変更が取り消される．このとき，実行が取り消された場合でも，あらかじめ支払った通貨は返金されないので，無駄なコストとなってしまう．

以降は，Tezos におけるガス消費量の計算について述べる．Tezos において，トップレベルのコードやラムダ，型などの値は全てバイトシーケンスとして保存され，送信される．コントラクトの実行において発生するガスの消費量は，以下の 8 つのコストに分けられていて，それぞれ計算方法や発生するタイミングが異なる．

1. データベースにアクセスして，必要とする値が存在するかどうか確認し，その値を読み込む．このとき，Reading Cost が発生する．
2. バイトシーケンスは，型なしの中間表現である Micheline 表現へと逆シリアル化される．このとき，Deserialization Cost が発生する．Micheline 表現では，全ての値が以下の要素で表される．

- integer
- string
- バイトシーケンス
- 命令や型などのプリミティブ
- 値のシーケンス

3. Micheline 表現は，プロトコル固有の型付き表現に解析される．このとき，Parsing Cost が発生する．
4. 型付き表現への解析において，ある型と別の型の等価性をチェックすることがある．このとき，Type Compaison Cost が発生する．
5. 型付き表現はインタプリタに渡され，コントラクトの内容が解釈実行される．このとき，Interpreter Cost が発生する．
6. コントラクトの実行後，型付き表現は Micheline 表現に変換される．このとき，Unparsing Cost が発生する．
7. Micheline 表現はバイトシーケンスへとシリアル化され，保存される．このとき，Serialization Cost が発生する．
8. コントラクトの実行によって変更されたデータをデータベースに書き込む．このとき，Writing Cost が発生する．

それぞれのコストは，以下に示すフィールドをもつレコードとして内部的に表現される．

{ allocations , steps , reads , writes , bytes_read , bytes_written }

各レコードには以下のように重みが設定されていて，コストに重みをかけることで，各コストをガスとして得ることができる．

allocation_weight = 2
step_weight = 1
read_base_weight = 100
write_base_weight = 160
byte_read_weight = 10
byte_written_weight = 15

例として，Reading Cost は通常，以下のようになる．

{ reads: scale 2
, bytes_read: scale $ <length of the value in bytes>
}

scale は値を実際に出力する値にスケーリングする関数のようなものである．このコストに重みをかけた結果として，ガス消費量が scale $ 200 + 10 * bytes_read と得られる．

2.3 Resource Aware ML について

Resource Aware ML(RAML) は，一階の関数プログラムの多項式リソース消費量の境界を，静的かつ自動的に計算する関数型プログラミング言語である．プログラムの文法は OCaml のものを採用しており，入力として OCaml の文法で書かれたプログラムを与えると，その多項式リソース境界を出力するツールとして扱うことができる．リソース消費量の分析は，ポテンシャルベースの償却解析によって行われる．後でかく

$$
\begin{aligned}
e \quad &::= \quad () \mid \mathrm{True} \mid \mathrm{False} \mid n \mid x \mid \\
&\qquad\quad x_1 \; binop \; x_2 \mid f(x_1, ..., x_n) \mid \\
&\qquad\quad \mathrm{let}\ x \ = \ e_1 \ \mathrm{in}\ e_2 \mid \mathrm{if}\ x\ \mathrm{then}\ e_t\ \mathrm{else}\ e_f \mid \\
&\qquad\quad (x_1, x_2) \mid \mathrm{match}\ x\ \mathrm{with}\ (x_1, x_2) \to e \mid \\
&\qquad\quad \mathrm{nil} \mid \mathrm{cons}(x_h, x_t) \mid \\
&\qquad\quad \mathrm{match}\ x\ \mathrm{with} \mid \mathrm{nil} \to e_1 \mid \mathrm{cons}(x_h, x_t) \to e_2 \mid \\
binop \quad &::= \quad + \mid - \mid * \mid \mathrm{mod} \mid \mathrm{div} \mid \mathrm{and} \mid \mathrm{or} \\
\mathrm{A} \quad &::= \quad \mathrm{unit} \mid \mathrm{bool} \mid \mathrm{int} \mid \mathrm{L(A)} \mid (\mathrm{A}, \mathrm{A}) \\
\mathrm{F} \quad &::= \quad (\mathrm{A}, ..., \mathrm{A}) \to \mathrm{A}
\end{aligned}
$$

図 1: RAML の文法

図 1 に RAML の文法を示す．e は RAML プログラム上の式を表していて，Unit 値 ()，Boolean 値 True，False，整数値 n，変数 x，変数 x_1 と x_2 の演算，関数適用，let 式を用いた局所関数を伴う式，if による条件分岐式，変数 x_1 と x_2 のペア，ペアに対する match 式，空リスト nil，リストの結合を表す cons(x_h, x_t)，リストに対する match 式がある．演算子 $binop$ には，整数値に対する演算である+，-，*，mod，div，Boolean 値に対する演算である and，or がある．A と F は，RAML プログラム上での simple type を表している．A はデータ型で，

Unit 型の unit，Boolean 型の bool，整数型の int，simple type の値のリスト，simple type の値のペアがある．F は関数型で，simple type の値を受け取って simple type の値を返す関数型を表している．また，RAML 上の well-typed な式を，この simple type が割り当てられた式と定義している．

RAML プログラムは，関数宣言のリストと main 式からなる．関数宣言は，関数の型宣言または関数の定義である．それぞれの関数定義に対して型宣言を行うことができるが，プログラム内で型宣言が行われていない関数については，プログラム実行時に型推論が行われる．main 式はリソース消費量の分析の対象となる式で，プログラムの最後に記述する．

RAML のリソース消費量の分析は，入力されたプログラムの，big-step operational semantics による各評価ステップに対して一定のコストを割り当てるメトリックによって，リソース消費量の計算を行う．メトリックは以下の 4 つが存在する．

- heap メトリックは，実行時に割り当てられたヒープセルの数を計算する．
- steps メトリックは，実行時の評価ステップ数を計算する．
- tick メトリックは，ユーザーが定義した tick 関数による tick 値を計算する．ユーザーは関数の定義中に Raml.tick(1.0) のような関数 (tick 関数) を定義することができる．tick 関数は呼び出される度に，引数の float 値に等しいリソース消費 (tick 値) が発生する．
- flips メトリックは，フリップ関数によるフリップ数を計算する．本論文では扱わないため，詳細な説明は省略する．

ユーザーは分析を行う際にメトリックを指定することで，自分の注目するリソースの消費量を分析の出力として得ることができる．

RAML プログラムの例として，リストに対するクイックソートを行うプログラムである quicksort のコードを Code 2 に示す．

```
let rec append l1 l2 =
  match l1 with
    | [] -> l2
    | x::xs -> x::(append xs l2)

let rec partition f l =
  match l with
```

```
8       | [] -> ([],[])
9       | x::xs ->
10        let (cs,bs) = partition f xs in
11        Raml.tick(1.0);
12        if f x then (cs,x::bs) else (x::cs,bs)
13
14  let rec quicksort gt = function
15    | [] -> []
16    | x::xs ->
17        let ys, zs = partition (gt x) xs in
18        append (quicksort gt ys) (x :: (quicksort gt zs))
19
20  let _ = quicksort (fun a b -> a <= b)  [9;8;7;6;5;4;3;2;1]
```

Code 2: quicksort.raml

見てわかる通り，プログラムのコードの見た目は OCaml に近いが，20 行目の let _ = ... の部分は OCaml には見られない表現である．この式が main 式で，リソース消費量の分析の対象となる．このプログラムは，append，partition，quicksort の 3 つの関数を定義し，main 式は quicksort の関数適用が記述されている．

1-4 行目の append 関数は，2 つのリスト l1,l2 を引数として受け取り，それらを結合したリストを返す関数である．6-12 行目の partition 関数は，リスト l と，l の要素の型の値を受け取って bool 型の値を返す関数 f を引数として受け取り，l の要素を f に関数適用したときの返り値によって 2 つのリストに分割する関数である．11 行目に tick 関数である Raml.tick(1.0) があり，結果として l の要素の数だけ 1.0 の tick 値が発生する．14-18 行目の quicksort 関数は，リスト l と，l の要素の型の値を 2 つ受け取って bool 型の値を返す関数 gt を引数として受け取り，l に対してクイックソートを実行する関数である．quicksort の定義中に append と partition が用いられている．

RAML のプログラムの実行において，主に evaluation と analysis の 2 つの操作がある．

evaluation は，プログラムの評価を行い，main 式の評価結果の返り値を出力する．また，先述した 4 つのメトリックによるリソース消費量を計算し出力する．evaluation は，./main eval [prog.raml] というコマンドによって実行され

る．ここで prog.raml は入力として用いるプログラムファイルである．

quicksort.raml を入力として evaluation を実行した結果を以下に示す．

```
1  $ ./main eval examples/quicksort.raml
2
3  Resource Aware ML, Version 1.5.0, June 2020
4
5  Typechecking expression ...
6    Typecheck successful.
7    Stack-based typecheck successful.
8
9  Evaluating expression ...
10
11   Return value:
12     [ 1; 2; 3; 4; 5; 6; 7; 8; 9 ]
13
14   Evaluation steps: 1624.00
15   Ticks:            36.00
16   Heap space:       547.00
17   Flips:            0.00
```

11-12 行目に，main 式の評価の返り値として，リスト [9;8;7;6;5;4;3;2;1] が正しくソートされた値が出力されている．また，14-17 行目に，上から steps，ticks，heap，flips と，それぞれのメトリックによって計算されたリソース消費量の値が出力されている．

analysis は，指定されたメトリックに則ってプログラムのリソース消費量の範囲の解析を実行する．解析の結果として，リソース消費量の範囲が，入力されたプログラムに依存する変数の多項式として出力される．出力される範囲は，リソース消費量の上限，下限，または上限と下限が一致した定数リソース境界から選ぶことができる．analysis は，./main analyze [mode] <metric> [<d1>] <d2> [-print (all | none | consume | level <lev>)] [-m] [prog.raml] [func_name] というコマンドで実行される．ここで，<>は指定必須のオプションで，[] は任意のオプションである．mode は出力される境界のタイプを upper，lower，constant から選ぶ．指定しない場合は upper となる．metric は分析に用いるメトリックを heap，steps，ticks，flips から選ぶ．d1 および d2 は，リソース消費量の境界の次数を指定する．分析は次数が d1,d1+1,...,d2 の範囲で行われ，出力される多項式もその範囲の次数となる．d1 を指定しない場合，d1=d2 として扱われる．-print はプログラムにおいて実行された関数の型を出力する．-print にもいくつかのオプションがあり，-print all は実行されたすべての関数の型を出力する．-print none は型の出力をしない．-print consume は消費関数（？）の型

を出力する．-print level <lev>は式を構文木として見た際に深さが<lev>以下の関数の型を出力する．-m は，指定するとモジュールモードとなり，main 式の代わりにトップレベルで定義された関数の型をすべて出力する．prog.raml は入力として用いるプログラムファイルである．func_name はモジュールモードでのみ指定することができるオプションで，指定した関数についてのみ型を出力する．

quicksort.raml を入力として，mode=upper，metric=steps，d1=1，d2=4，-print level 1 とオプションを設定して analysis を実行した結果を以下に示す．

```
1  $ ./main analyze steps 1 4 -print level 1 examples/quicksort.raml
2
3  Resource Aware ML, Version 1.5.0, June 2020
4
5  Typechecking expression ...
6    Typecheck successful.
7    Stack-based typecheck successful.
8
9  Analyzing expression ...
10
11   Trying degree: 1, 2
12
13   Function types:
14
15 == quicksort :
16
17   [int -> int -> bool; int list] -> int list
18
19   Non-zero annotations of the argument:
20         35  <--  (*, [::(*); ::(*)])
21         36  <--  (*, [::(*)])
22          3  <--  (*, [])
23
24   Non-zero annotations of result:
25
26   Simplified bound:
27      3 + 18.5*M + 17.5*M^2
28    where
29      M is the number of ::-nodes of the 2nd component of the argument
30
31 ====
32
33   Derived upper bound: 1624.00
34
35   Mode:           upper
36   Metric:         steps
37   Degree:         2
38   Run time:       0.14 seconds
39   #Constraints:  638
```

11 行目に Trying degree: 1,2 とあるが，指定された次数の 1,2,3,4 の低い値か

ら順に分析を行い，次数が 2 のときに分析が成功したことを示している．指定された次数において分析が成功しない場合，その旨がエラーメッセージで表示される．15-29 行目には，main 式で用いられた関数 quicksort の分析の結果が示されている．17 行目に quicksort の型が示されている．[] 中の型が引数の型で，複数ある場合は; で区切られている．19-22 行目に，引数のポテンシャル注釈が引数のデータ構造ごとに示されている．このポテンシャル注釈に関する情報を多項式に変換したものが，26-29 行目に示されている．そして，33 行目に main 式の上界の値が出力され，35-39 行目に分析のオプションや計算時間，計算量が出力されている．

なお，先述したように RAML の文法は OCaml のものを採用しているが，RAML においてサポートされていない OCaml の機能がある．以下にその例を示す．

- オブジェクト指向言語としての特徴
- モジュール
- 複雑な帰納的データ型
- 文字列や文字

3 RAML での Michelson プログラムの実装

本節では，RAML を用いて，Michelson プログラムの挙動を模倣するプログラムを実装する手法について述べる．具体的には，Michelson において実装されている文法や命令を模倣するライブラリを RAML において設計する，また，設計したライブラリについて，各命令において発生するガス消費量を表す tick 関数を定義する．このライブラリを用いて，Michelson プログラムの挙動を模倣する RAML プログラムを実装する．第 3.1 節では文法について，第 3.2 節では命令について，第 3.3 節ではライブラリを用いたプログラムについて，第 3.4 節ではガス消費量に関する tick 関数の定義について述べる．なお，

3.1 文法

Michelson の文法は，すでに第 2.1 節で示した．Michelson の文法をライブラリで設計するにあたって，スタックの要素を OCaml のヴァリアント型を用いて表す．Code3 に実装したヴァリアント型 t を示す．

```
type t =
  Int of int | Nat of Rnat.t | Mutez of Rnat.t |
  Bool of bool | Address | Unit | MNone | MSome of t |
  LNil | LCons of t * t | Operation | Contract |
  Pair of t * t | Left of t | Right of t
```

Code 3: スタックの要素を表すヴァリアント型

ヴァリアント型 t のコンストラクタとして，int 型の Int，nat 型の Nat，mutez 型の Mutez，bool 型の Bool，address 型の Address，unit 型の Unit，option 型の MNone，MSone，list 型の LNil，LCons，operation 型の Operation，contract 型の Contract，pair 型の Pair，or 型の Left，Right がある．Nat の引数の型である Rnat.t は，RAML に用意されている自然整数を表す型で，四則演算と，n が 0 か 1 以上かで分岐する条件分岐関数が用意されている．Operation，Address，Contract については，簡略化のために引数をとらない単一のコンストラクタとして扱う．また，list 型の要素を設計するにあたって，先述したように RAML において複雑な帰納的データ型の実装には制限があり，ヴァリアント型 t のコンストラクタとして，t list 型の値を引数にとるコンストラクタを宣言することができない．その代替として，空リストを表す LNil と，リストの要素と元のリストを引数としてリストの結合を表す LCons をコンストラクタとして宣言する．なお，このヴァリアント型の設計上，LCons の 2 つ目の引数は型 t の値であればよい，とされているが，コントラクトを模倣するプログラムの実行において，LCons の 2 つ目の引数が LNil または LCons となるように命令の関数が定義されている．

このヴァリアント型 t を用いて，スタックを型 t のリストとして設計することができる．

3.2 命令

Michelson の命令は，スタックを受け取り，そのスタックの内容を書き換える操作として実装されている．この挙動を模倣する関数を，第 3.1 節で設計した文法において，スタックに相当するリストを受け取って，書き換え後のスタックを返すような (t list->t list) 型をもつ関数として定義する．

以下，実装した関数について説明するが，適宜付録の RAML ライブラリの

ソースコードを行数を示して参照する．

- 6 行目の failwith は FAILWITH 命令に相当する関数で，例外 Invalid_argument を発生させる関数として定義される．
- 8 行目の nop は受け取ったスタックをそのまま返す関数である．IF 命令などの分岐命令の引数に用いる場合がある．
- 10 行目の p は 1.0 の tick 値を発生させる関数で，Michelson プログラムでの {，} をこれに置換する．後述する tick 関数によるガス消費量の見積もりにおいて用いる．
- 12 行目の中置演算子 |>は，引数として受け取った 2 つの関数 f，g を，引数のスタックに対して続けて適用する演算子である．Michelson における命令のシーケンスに相当する役割をもつ．
- 14-17 行目の if_は IF 命令に相当する関数で，スタックのトップが Bool b ならば，b の値に応じて引数の命令シーケンス bt，bf のいずれかを残ったスタックに対して適用する関数として定義されている．
- 19-28 行目の loop は LOOP 命令に相当する関数であるが，Michelson の LOOP の挙動を再現すると解析が難しくなるため，挙動を簡単なものにしている．具体的には，引数として Rnat.t 型の引数 n を受け取って，スタックのトップに関わらず，引数の命令シーケンスを n 回残りのスタックに適用する．20 行目の Rnat.ifz は，1 つ目の引数の Rnat.t 型の値 n が 0 ならば 2 つ目の引数の関数を，n が 1 以上ならば n′=n-1 として 3 つ目の引数の関数を適用する条件分岐関数である．
- 30-33 行目の dip1 は DIP 命令に相当する関数で，スタックのトップを保持して，残りのスタックに対して引数の命令のシーケンスを適用する関数として定義される．
- 35-38 行目の drop1 は DROP 命令に相当する関数で，スタックのトップを取り除く関数として定義される．
- 40-43 行目の dup は DUP 命令に相当する関数で，スタックのトップの要素を複製してスタックに積む関数として定義される．
- 45-48 行目の swap は SWAP 命令に相当する関数で，スタックのトップと 2 番目の要素を入れ替える関数として定義される．
- 50 行目の push は PUSH 命令に相当する関数で，受け取った要素をスタックに積む命令として定義されている．なお，スタックの要素の型の指定は

なく，値のみを引数として受け取る．

- 52 行目の unit は UNIT 命令に相当する関数で，Unit をスタックに積む命令として定義される．
- 54-82 行目の eq，neq，lt，gt，le，ge は，それぞれ EQ，NEQ，LT，GT，LE，GE 命令に相当する関数で，スタックのトップが Int i ならば，i と 0 の比較を行い，その結果を表す Bool をスタックに積む関数として定義されている．
- 84-97 行目の or_，and_，xor は OR，AND，XOR 命令に相当する関数で，スタックのトップと 2 つ目の要素がともに Bool ならば，2 つの Boolean 値の論理演算を行い，その結果の Bool をスタックに積む関数として定義される．
- 99-102 行目の not_は NOT 命令に相当する関数で，スタックのトップが Bool b ならば，b の論理否定の Bool をスタックに積む関数として定義されている．
- 104-108 行目の neg は NEG 命令に相当する関数で，スタックのトップが Int または Nat ならば，その値の正負を反転した値の Int をスタックに積む関数として定義される．
- 110-114 行目の abs は ABS 命令に相当する関数で，スタックのトップが Int i ならば，i の絶対値の Nat をスタックに積む関数として定義される．
- 116-120 行目の isnat が ISNAT 命令に相当する関数で，スタックのトップの要素が Int i ならば，i の正負に応じて，Nat のオプション値をスタックに積む関数として定義される．
- 122-125 行目の int は INT 命令に相当する関数で，スタックのトップが Nat ならば，それを Int に変換する関数として定義される．
- 127-178 行目の add，sub，mul，ediv は，それぞれ ADD，SUB，MUL，EDIV 命令に相当する関数で，スタックのトップと 2 番目の要素の四則演算を行う関数である．スタックのトップと 2 番目の要素の型のパターンに応じてスタックに積む要素の型が決まるので，パターンマッチングによって分ける．また，ediv ではスタックの 2 番目の要素が 0 かそうでないかで条件分岐があり，スタックに積む要素の型は値のペアのオプション型となる．
- 180-197 行目の compare は COMPARE 命令に相当する関数で，スタックのトップと 2 番目の要素の比較を行い，トップの要素の方が大きい場合は Int 1 を，2 番目の要素の方が大きい場合は Int -1 を，等しい場合は Int 0 をス

タックに積む関数である．Int，Nat，Mutez について比較を行うことができるが，違う型同士の比較はできない．

- 199-202 行目の pair は PAIR 命令に相当する関数で，スタックのトップと 2 番目の要素をペアにしてスタックに積む関数として定義される．
- 204-212 行目の car，cdr は，それぞれ CAR，CDR 命令に相当する関数で，スタックのトップの要素がペアならば，ペアの第 1 要素，もしくは第 2 要素をスタックに積む関数として定義される．
- 214-217 行目の some は SOME 命令に相当する関数で，スタックのトップの要素をオプション値 MSone に渡してスタックに積む関数として定義される．
- 219 行目の none は NONE 命令に相当する関数で，オプション値 MNone をスタックに積む関数として定義される．
- 225 行目の if_none は IF_NONE 命令に相当する関数で，スタックのトップが MNone ならば 1 つ目の引数の命令シーケンスを，スタックのトップが MSome なら 2 つ目の引数の命令シーケンスを残りのスタックに適用する関数として定義される．
- 227-235 行目の left，right はそれぞれ LEFT，RIGHT 命令に相当する関数で，スタックのトップの要素をユニオンの Left，Right に渡してスタックに積む関数として定義される．
- 237-241 行目の if_left は IF_LEFT 命令に相当する関数で，スタックのトップが Left ならば 1 つ目の引数の命令シーケンスを，スタックのトップが Right なら 2 つ目の引数の命令シーケンスを残りのスタックに適用する関数として定義される．
- 243-247 行目の cons は CONS 命令に相当する関数で，スタックの 2 番目の要素が LCons ならば，スタックのトップの要素と結合したリストをスタックに積む関数として定義される．
- 249 行目の nil は NIL 命令に相当する関数で，LNil をスタックに積む関数として定義される．
- 251-255 行目の if_cons は IF_CONS 命令に相当する関数で，スタックのトップの要素が LCons ならば 1 つ目の引数の命令シーケンスを，スタックの要素が LNil ならば 2 つ目の引数の命令シーケンスを残りのスタックに適用する関数として定義される．

- 257-279 行目の map_list は MAP 命令に相当する関数で，スタックのトップがリストならば，リストに対する命令シーケンスのマッピングを行う関数である．map_list_aux はこれの補助関数である．
- 281-291 行目の size_list は SIZE 命令に相当する関数で，スタックのトップがリストならば，リストの長さの Nat をスタックに積む関数である．size_list_aux はこれの補助関数である．
- 293-303 行目の iter_list は ITER 命令に相当する関数で，スタックのトップがリストならば，リストに対する命令シーケンスのイテレートを行う関数である．iter_list_aux はこれの補助関数である．
- 305-308 行目の transfer_tokens は TRANSFER_TOKENS 命令に相当する関数で，スタックの要素がトップからパラメータ p，通貨量 m，コントラクト c となっているならば，c に対して m を p と共にを送る操作をスタックに積むという命令を，引数などを簡略化した上で関数として定義されている．
- 310-313 行目の contarct は CONTRACT 命令に相当する関数で，スタックのトップが Address ならば，MSome Contract をスタックに積む関数として定義される．本来は型を引数で受け取り，スタックのトップのアドレスがその型のコントラクトに関連づけられているかどうかを検査するが，この関数においては簡略化されている．
- 315 行目の source，317 行目の amount は，それぞれ SOURCE，AMOUNT 命令に相当する関数で，対応する Address または Mutez をスタックに積む関数である．積まれるアドレスや通貨量の情報については簡略化されている．

3.3 ライブラリを用いたプログラム

Michelson のプログラムでは，プログラムに与えられる parameter と，ブロックチェーン上に保存されている storage の型宣言から始まり，parameter と storage のペアが積まれた初期スタックに対して，一連の命令が順番に実行される．このプログラムの挙動を模倣する RAML のプログラムを，設計したライブラリを用いて，初期スタックのリストに対して関数を順に適用するプログラムとして実装する．

Code1 に示したプログラム example1 の挙動を模倣する RAML プログラムは

以下のようになる．

```
let _ =
  (pair
  (nil
  (add
  (dip1 (p |> cdr |> p)
  (car
  (dup
  (car
  (Pair (Pair (Int 3, Int 5),  Int 0) :: [])))))))))
```

プログラムの設計上，Michelson プログラムと記述の順番が逆になっていて読みにくいことをご了承されたい．Michelson プログラムでは parameter と storage の型宣言から始まるが，RAML プログラムでは初期スタックの値を宣言し，それに対して命令に相当する関数を順に適用する．5 行目の dip1 の引数となる命令シーケンスは，命令と命令の間を |>で繋ぐことによって表す．また，シーケンスの最初と最後は {，} に相当する p を記述する．

このプログラムの evaluation を実行した結果を以下に示す．

```
$ ./main eval michelson/example1.raml

Resource Aware ML, Version 1.5.0, June 2020

Typechecking expression ...
  Typecheck successful.
  Stack-based typecheck successful.

Evaluating expression ...

  Return value:
    [ Pair ( (), LNil (), Int 8 )
                                              ]

  Evaluation steps: 270.00
  Ticks:            35.00
  Heap space:       166.00
  Flips:            0.00
```

11-13 行目に，操作のリストと，parameter として受け取った整数値の和のペアが入ったスタックが返り値として示されている．

3.4 tick メトリックによるガス消費量の見積もり

スマートコントラクトのガス消費量を見積もりを行うにあたって，コントラクトを実際に実行してそのガス消費量を調べる必要がある．コントラクトの実行環境として，Tezos ハンズオン [1)]のサンドボックス環境を用いる．この環境を利用する利点として，単体のコンピュータで完結していて外部のネットワークを必要としていない点，環境をいつでもリセットできる点が挙げられる．

第 2.2 節でも述べたように，コントラクトの実行において発生するガスの消費量は 8 つのコストに分けられていて，それぞれ計算方法が異なる．ゆえに，コントラクトの実行コスト全体を見積もることは難しいと判断し，8 つのコストのうちの一つである Interpreter Cost の見積もりを行うことにした．interpreter cost を選んだ理由として，コストとしての大きさが他のコストより小さいこと，命令毎にコストが設定されているので，後述する方法による見積もりがしやすいことが挙げられる．

コントラクトの Interpreter Cost を調べるにあたって，run script <src> on storage <storage> and input <input> –trace-stack [–gas <gas>] というコマンドを実行し，ガス消費の推移を調べるという方法を用いる．これは，プログラムファイル<src>のスクリプトを，parameter を<input>，storage を<storage>とした上で実行するコマンドで，–trace-stack というオプションをつけることで 1 ステップ毎のスタックの状態，ガスの残量（使用許容量-消費量）が出力される．–gas <gas>はスクリプト開始時のガスの使用許容量を指定できるオプションである．

Code1 に示したプログラム example1 について run script を実行した結果を以下に示す．

```
1  $ ./tezos-client run script contracts/pairadd.tz on storage 0 and input 'Pair 3 5'
     --trace-stack --gas 100000
2  storage
3    8
4  emitted operations
5
6  trace
7    - location: 8 (remaining gas: 99655 units remaining)
8      [ (Pair (Pair 3 5) 0)    ]
9    - location: 9 (remaining gas: 99652 units remaining)
10     [ (Pair 3 5)    @parameter ]
11   - location: 10 (remaining gas: 99649 units remaining)
```

[1)] https://gitlab.com/dailambda/docker-tezos-hands-on

```
12      [ (Pair 3 5)    @parameter
13        (Pair 3 5)    @parameter ]
14    - location: 11 (remaining gas: 99646 units remaining)
15      [ 3
16        (Pair 3 5)    @parameter ]
17    - location: 14 (remaining gas: 99640 units remaining)
18      [ 5       ]
19    - location: 13 (remaining gas: 99639 units remaining)
20      [ 5       ]
21    - location: 12 (remaining gas: 99639 units remaining)
22      [ 3
23        5       ]
24    - location: 15 (remaining gas: 99626 units remaining)
25      [ 8       ]
26    - location: 16 (remaining gas: 99620 units remaining)
27      [ {}
28        8       ]
29    - location: 18 (remaining gas: 99612 units remaining)
30      [ (Pair {} 8)     ]
31    - location: -1 (remaining gas: 99611 units remaining)
32      [ (Pair {} 8)     ]
```

6 行目の trace 以降の出力で，1 ステップ毎のスタックの状況，ガス残量が示されている．開始時のガス許容量を 100000units と指定していてるが，最初の出力でいくらか減っていることから，Reading Cost，Deserialization Cost，Parsing Cost，Type Comparison Cost が消費された時点のガス残量が示されていると考えられる．このガス残量から，最後の出力時点でのガス残量を引いた値が Interpreter Cost であると考えられる．trace 以降を見てみると，location はプログラムの実行の進行度合いを示す値で，例えば 7-10 行目を見てみると，location が 8 から 9 になり，最初の命令である CAR が実行されてスタックが書き換えられ，ガスが 3units 消費されていることがわかる．ここから CAR 命令の Interpreter Cost が 3units であると推測できる．また，17-23 行目では DIP {CDR} が実行されているが，命令のシーケンスなどにおいて{，}を読み込むとガスが 1units 消費されていることがわかる．これが Interpreter Cost に含まれるかどうかは定かではないが，この消費量も含めて見積もることとする．以上の例に倣って，様々な Michelson プログラムについて run script コマンドを実行し，各命令の Interpreter Cost を調べた．

見積もりの方法として，tick メトリックを用いた解析を用いる．第 3.2 節で実装した各命令について，その命令の Interpreter Cost に相当する値の tick 関数を定義し，ライブラリを用いて実装したプログラムを tick メトリックを用いて解析し，解析の結果として出力される tick 値の上界を Interpreter Cost の見積

もり結果と見なす.

各命令に定義した tick 関数については，付録の RAML ライブラリのソースコードを参照されたい．以下，tick 関数によって正確に Interpreter Cost を見積もることのできない命令について述べる．

- ADD，SUB，MUL，EDIV の整数値の演算の命令における Interpreter Cost は，演算の対象となる整数値の絶対値によって変動し [1)]，それを tick 関数で表すことはできなかった．ライブラリの命令においては，スタックの要素にに関わらず一定の tick 値を発生するように定義されている．
- CONTRACT 命令の Interpreter Cost は，例外的に 10000units 以上となっており，さらに対象となるアドレスによって増減するため，設計したライブラリにおいて見積もるには情報が不十分であった．ライブラリの命令においては，tick 関数を定義していない．

4　ガス消費量の解析の結果と考察

5　改善点

6　結論

謝辞

参考文献

[1] Caplener, H. D. and Janku, J. A.: Improved Modeling of Computer Hardware Systems, *Computer Design*, Vol. 12, pp. 59–64 (1973).

[2] Beizer, B.: Towards a New Theory of Sequential Switching Networks, *IEEE Trans. Computers*, Vol. C–19, pp. 936–956 (1970).

[3] 村上伸一: 微分方程式の解曲線の表示, 情報処理, Vol. 14, pp. 231–238 (1970).

[4] 平井有三, 福島邦彦: 両眼視差抽出機構の神経回路網モデル, 信学論 (D), Vol. 56–D, pp. 465–472 (1973).

[5] Baraff, D.: Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation, *SIGGRAPH '90 Proceedings* (Beach, R. J.(ed.)), Dallas,

[1)] 2 つの整数値の絶対値のうち大きい方を n とすると，$\log_2 n$ に比例する．

Texas, ACM, Addison-Wesley, pp. 19–28 (1990).

[6] 對馬雄次ほか: ボリュームレンダリング専用並列計算機のアーキテクチャ, 並列処理シンポジウム JSPP'94, pp. 89–96 (1994).

[7] Barnett, S. and Storey, C.: *Matrix Methods in Stablity Theory*, Nelson, London (1970).

[8] J. E. ホップクロフト, J. D. ウルマン（木村, 野崎訳）: 言語理論とオートマトン, サイエンス社, chapter 6 (1972).

[9] 寺沢寛一: 自然科学者のための数学概論, 岩波書店, pp. 325–328 (1955).

付録： 本研究において実装した RAML ライブラリのソースコード

```
exception Invalid_argument

type t =
  Int of int | Nat of Rnat.t | Mutez of Rnat.t | Bool of bool
    | Address | Unit | MNone | MSome of t | LNil | LCons of
   t * t | Operation | Contract | Pair of t * t | Left of t
   | Right of t

let failwith s = raise Invalid_argument

let nop s = s

let p = fun s -> Raml.tick(1.0); s

let (|>) f g = fun s -> g (f s)

let if_ (bt:t list -> t list) (bf:t list -> t list) s =
  match s with
  | Bool b :: xs -> Raml.tick(3.0); if b then bt xs else bf
    xs
  | _ ->  raise Invalid_argument

let rec loop n body s =
  Rnat.ifz n
    (fun () ->
      match s with
      | _ :: xs -> Raml.tick(2.0); xs
      | _ -> raise Invalid_argument)
    (fun n' ->
      match s with
      | _ :: xs -> Raml.tick(2.0); loop n' body (body xs)
      | _ -> raise Invalid_argument)

let dip1 body s =
  match s with
  | x :: xs -> Raml.tick(2.0); x :: (body xs)
```

```
33    | _ -> raise Invalid_argument
34
35  let drop1 s =
36    match s with
37    | _ :: xs -> Raml.tick(3.0); xs
38    | _ -> raise Invalid_argument
39
40  let dup s =
41    match s with
42    | x :: xs -> Raml.tick(3.0); x :: x :: xs
43    | _ -> raise Invalid_argument
44
45  let swap s =
46    match s with
47    | x :: y :: xs -> Raml.tick(3.0); y :: x :: xs
48    | _ -> raise Invalid_argument
49
50  let push elm s = Raml.tick(3.0); elm :: s
51
52  let unit s = Raml.tick(3.0); Unit :: s
53
54  let eq s =
55    match s with
56    | Int i :: xs -> Raml.tick(3.0); Bool (i = 0) :: xs
57    | _ -> raise Invalid_argument
58
59  let neq s =
60    match s with
61    | Int i :: xs -> Raml.tick(3.0); Bool (not (i = 0)) :: xs
62    | _ -> raise Invalid_argument
63
64  let lt s =
65    match s with
66    | Int i :: xs -> Raml.tick(3.0); Bool (i < 0) :: xs
67    | _ -> raise Invalid_argument
68
69  let gt s =
70    match s with
71    | Int i :: xs -> Raml.tick(3.0); Bool (i > 0) :: xs
```

```
72    | _ -> raise Invalid_argument
73
74  let le s =
75    match s with
76    | Int i :: xs -> Raml.tick(3.0); Bool (i <= 0) :: xs
77    | _ -> raise Invalid_argument
78
79  let ge s =
80    match s with
81    | Int i :: xs -> Raml.tick(3.0); Bool (i >= 0) :: xs
82    | _ -> raise Invalid_argument
83
84  let or_ s =
85    match s with
86    | Bool bx :: Bool by :: xs -> Raml.tick(3.0); Bool (bx ||
       by) :: xs
87    | _ -> raise Invalid_argument
88
89  let and_ s =
90    match s with
91    | Bool bx :: Bool by :: xs -> Raml.tick(3.0); Bool (bx &&
       by) :: xs
92    | _ -> raise Invalid_argument
93
94  let xor s =
95    match s with
96    | Bool bx :: Bool by :: xs -> Raml.tick(3.0); Bool ((bx ||
       by) && (not bx || not by)) :: xs
97    | _ -> raise Invalid_argument
98
99  let not_ s =
100   match s with
101   | Bool b :: xs -> Raml.tick(3.0); Bool (not b) :: xs
102   | _ ->  raise Invalid_argument
103
104 let neg s =
105   match s with
106   | Int i :: xs -> Raml.tick(6.0); Int (-i) :: xs
```

```
107   | Nat n :: xs -> Raml.tick(6.0); Int (-(Rnat.to_int n)) ::
      xs
108   | _ -> raise Invalid_argument
109
110 let abs s =
111   match s with
112   | Int i :: xs -> Raml.tick(6.0);
113     if i >= 0 then Nat (Rnat.of_int i) :: xs else Nat (Rnat.
        of_int (-i)) :: xs
114   | _ -> raise Invalid_argument
115
116 let isnat s =
117   match s with
118   | Int i :: xs -> Raml.tick(6.0);
119     if i >= 0 then MSome (Nat (Rnat.of_int i)) :: xs else
        MNone :: xs
120   | _ -> raise Invalid_argument
121
122 let int s =
123   match s with
124   | Nat n :: xs -> Raml.tick(3.0); Int (Rnat.to_int n) :: xs
125   | _ -> raise Invalid_argument
126
127 let add s =
128   match s with
129   | Int ix :: Int iy :: xs -> Raml.tick(4.0); Int (ix + iy)
      :: xs
130   | Int ix :: Nat iy :: xs -> Raml.tick(4.0); Int (ix + (Rnat
      .to_int iy)) :: xs
131   | Nat ix :: Int iy :: xs -> Raml.tick(4.0); Int ((Rnat.
      to_int ix) + iy) :: xs
132   | Nat ix :: Nat iy :: xs -> Raml.tick(4.0); Nat (Rnat.add
      ix iy) :: xs
133   | Mutez ix :: Mutez iy :: xs -> Raml.tick(7.0); Mutez (Rnat
      .add ix iy) :: xs
134   | _ -> raise Invalid_argument
135
136 let sub s =
137   match s with
```

```
138   | Int ix :: Int iy :: xs -> Raml.tick(4.0); Int (ix - iy)
      :: xs
139   | Int ix :: Nat iy :: xs -> Raml.tick(4.0); Int (ix - (Rnat
      .to_int iy)) :: xs
140   | Nat ix :: Int iy :: xs -> Raml.tick(4.0); Int ((Rnat.
      to_int ix) - iy) :: xs
141   | Nat ix :: Nat iy :: xs -> Raml.tick(4.0); Int ((Rnat.
      to_int ix) - (Rnat.to_int iy)) :: xs
142   | Mutez ix :: Mutez iy :: xs -> Raml.tick(7.0);
143     let (m, _) = Rnat.minus ix iy in Mutez m :: xs
144   | _ -> raise Invalid_argument
145
146 let mul s =
147   match s with
148   | Int ix :: Int iy :: xs -> Raml.tick(4.0); Int (ix * iy)
      :: xs
149   | Int ix :: Nat iy :: xs -> Raml.tick(4.0); Int (ix * (Rnat
      .to_int iy)) :: xs
150   | Nat ix :: Int iy :: xs -> Raml.tick(4.0); Int ((Rnat.
      to_int ix) * iy) :: xs
151   | Nat ix :: Nat iy :: xs -> Raml.tick(4.0); Nat (Rnat.mult
      ix iy) :: xs
152   | Mutez ix :: Nat iy :: xs -> Raml.tick(13.0); Mutez (Rnat.
      mult ix iy) :: xs
153   | Nat ix :: Mutez iy :: xs -> Raml.tick(13.0); Mutez (Rnat.
      mult ix iy) :: xs
154   | _ -> raise Invalid_argument
155
156 let ediv s =
157   match s with
158   | Int ix :: Int iy :: xs -> Raml.tick(10.0);
159     if iy = 0 then MNone :: xs else MSome (Pair (Int (ix / iy
        ), Nat (Rnat.of_int (ix mod iy)))) :: xs
160   | Int ix :: Nat iy :: xs -> Raml.tick(10.0);
161     Rnat.ifz iy
162       (fun () -> MNone :: xs)
163       (fun n' -> MSome (Pair (Int (ix / (Rnat.to_int iy)),
          Nat (Rnat.of_int (ix mod (Rnat.to_int iy))))) :: xs)
164   | Nat ix :: Int iy :: xs -> Raml.tick(10.0);
```

```
165     if iy = 0 then MNone :: xs else MSome (Pair (Int ((Rnat.
          to_int ix) / iy), Nat (Rnat.of_int ((Rnat.to_int ix)
          mod iy)))) :: xs
166   | Nat ix :: Nat iy :: xs -> Raml.tick(10.0);
167     Rnat.ifz iy
168       (fun () -> MNone :: xs)
169       (fun n' -> let (d, m, _) = Rnat.div_mod ix iy in MSome
            (Pair (Nat d, Nat m)) :: xs)
170   | Mutez ix :: Nat iy :: xs -> Raml.tick(16.0);
171     Rnat.ifz iy
172       (fun () -> MNone :: xs)
173       (fun n' -> let (d, m, _) = Rnat.div_mod ix iy in MSome
            (Pair (Mutez d, Mutez m)) :: xs)
174   | Mutez ix :: Mutez iy :: xs -> Raml.tick(22.0);
175     Rnat.ifz iy
176       (fun () -> MNone :: xs)
177       (fun n' -> let (d, m, _) = Rnat.div_mod ix iy in MSome
            (Pair (Nat d, Mutez m)) :: xs)
178   | _ -> raise Invalid_argument
179
180 let rec compare s =
181   match s with
182   | Int ix :: Int iy :: xs -> Raml.tick(4.0); begin
183     if ix < iy then Int (-1) :: xs else begin
184       if ix = iy then Int 0 :: xs else Int 1 :: xs
185     end
186   end
187   | Nat ix :: Nat iy :: xs -> Raml.tick(4.0); begin
188     if Rnat.to_int ix < Rnat.to_int iy then Int (-1) :: xs
        else begin
189       if Rnat.to_int ix = Rnat.to_int iy then Int 0 :: xs
          else Int 1 :: xs
190     end
191   end
192   | Mutez ix :: Mutez iy :: xs -> Raml.tick(3.0); begin
193     if Rnat.to_int ix < Rnat.to_int iy then Int (-1) :: xs
        else begin
194       if Rnat.to_int ix = Rnat.to_int iy then Int 0 :: xs
          else Int 1 :: xs
```

```
195       end
196     end
197     | _ -> raise Invalid_argument
198
199 let pair s =
200     match s with
201     | x :: y :: xs -> Raml.tick(8.0); (Pair (x, y)) :: xs
202     | _ -> raise Invalid_argument
203
204 let car s =
205     match s with
206     | Pair (a, _) :: xs -> Raml.tick(3.0); a :: xs
207     | _ -> raise Invalid_argument
208
209 let cdr s =
210     match s with
211     | Pair (_, b) :: xs -> Raml.tick(3.0); b :: xs
212     | _ -> raise Invalid_argument
213
214 let some s =
215     match s with
216     | x :: xs -> Raml.tick(6.0); MSome x :: xs
217     | _ -> raise Invalid_argument
218
219 let none s = Raml.tick(6.0); MNone :: s
220
221 let if_none bt bf s =
222     match s with
223     | MNone :: xs -> Raml.tick(5.0); bt xs
224     | MSome x :: xs -> Raml.tick(5.0); bf (x :: xs)
225     | _ -> raise Invalid_argument
226
227 let left s =
228     match s with
229     | x :: xs -> Raml.tick(6.0); Left x :: xs
230     | _ -> raise Invalid_argument
231
232 let right s =
233     match s with
```

```
234   | x :: xs -> Raml.tick(6.0); Right x :: xs
235   | _ -> raise Invalid_argument
236
237 let if_left bt bf s =
238   match s with
239   | Left a :: xs -> Raml.tick(5.0); bt (a :: xs)
240   | Right b :: xs -> Raml.tick(5.0); bf (b :: xs)
241   | _ -> raise Invalid_argument
242
243 let cons s =
244   match s with
245   | x :: LCons(y, z) :: xs -> Raml.tick(8.0); LCons(x, LCons(
      y, z)) :: xs
246   | x :: LNil :: xs -> Raml.tick(8.0); LCons(x, LNil) :: xs
247   | _ -> raise Invalid_argument
248
249 let nil s = Raml.tick(6.0); LNil :: s
250
251 let if_cons bt bf s =
252   match s with
253   | LNil :: xs -> Raml.tick(5.0); bf xs
254   | LCons (hd, tl) :: xs -> Raml.tick(5.0); bt (hd :: tl ::
      xs)
255   | _ -> raise Invalid_argument
256
257   let rec map_list_aux body hd tl s =
258   match tl with
259   | LNil -> begin
260     match (body (hd :: s)) with
261     | [] -> raise Invalid_argument
262     | x' :: xs' -> Raml.tick(3.0); LCons (x', LNil) :: xs'
263   end
264   | LCons (hd', tl') -> begin
265     match (body (hd :: s)) with
266     | [] -> raise Invalid_argument
267     | x' :: xs' -> Raml.tick(1.0); begin
268       match (map_list_aux body hd' tl' xs') with
269       | [] -> raise Invalid_argument
270       | x'' :: xs'' -> LCons (x', x'') :: xs''
```

```
271     end
272   end
273   | _ -> raise Invalid_argument
274
275 let map_list body s =
276   match s with
277   | LNil :: xs -> Raml.tick(4.0); s
278   | LCons (hd, tl) :: xs -> Raml.tick(2.0); map_list_aux body
        hd tl xs
279   | _ -> raise Invalid_argument
280
281 let rec size_list_aux tl =
282   match tl with
283   | LNil -> Rnat.zero
284   | LCons (_, tl') -> Rnat.add (Rnat.of_int 1) (size_list_aux
        tl')
285   | _ -> raise Invalid_argument
286
287 let size_list s =
288   match s with
289   | LNil :: xs -> Raml.tick(4.0); Nat (Rnat.zero) :: xs
290   | LCons (_, tl) :: xs -> Raml.tick(4.0); Nat (Rnat.add (
      Rnat.of_int 1) (size_list_aux tl)) :: xs
291   | _ -> raise Invalid_argument
292
293 let rec iter_list_aux body hd tl s =
294   match tl with
295   | LNil -> Raml.tick(3.0); body (hd :: s)
296   | LCons (hd', tl') -> Raml.tick(1.0); iter_list_aux body hd
      ' tl' (body (hd :: s))
297   | _ -> raise Invalid_argument
298
299 let iter_list body s =
300   match s with
301   | LNil :: xs -> Raml.tick(4.0); xs
302   | LCons (hd, tl) :: xs -> Raml.tick(2.0); iter_list_aux
      body hd tl xs
303   | _ -> raise Invalid_argument
304
```

```
let transfer_tokens s =
  match s with
  | _ :: Mutez _ :: Contract :: xs -> Raml.tick(20.0);
    Operation :: xs
  | _ -> raise Invalid_argument

let contract s =
  match s with
  | Address :: xs ->  if true then MSome Contract :: xs else
    MNone :: xs
  | _ -> raise Invalid_argument

let source s = Raml.tick(3.0); Address :: s

let amount s = Raml.tick(3.0); Mutez Rnat.zero :: s
```