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．このような実行コストの無駄をを抑えるために，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手法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 で記述される．このプログラムを RAML で解析するために，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 で実装する．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，プログラムのガス消

費量を見積もることができる．

Michelsonの命令は受け取ったスタックを書き換える．この挙動をRAMLで再現するために，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する．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．こ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．

Tezosのコントラクトの実行に伴うガスの消費量のうち，本研究では特に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とする．各命令のガス消費量はRAMLのtickメトリックを用いて表現する．tickメトリックは，リソースの消費量や発生するタイミングをユーザーが定義するためのメトリックで，RAMLに備えられている関数を呼び出すことでその関数のリソース消費量を定義することができる．本ライブラリの各命令について，その命令のinterpreter costに相当する値の関数を呼び出すよう定義し，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，ガス消費量がスタック中の値に依存す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．

本研究において実装しなかった命令や，解析が正しく行えなかった命令を扱えるようにライブラリを拡張することは，今後の課題である．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く．

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. Many cryptocurrencies support *smart contracts*, which are programs executed on a blockchain. A smart contract is used to execute a transaction involving cryptocurrencies automatically.

There is a concept of *gas consumption* in smart contracts. Gas is the fee for the computational resources used to execute a contract. When a contract is executed, the execution of each instruction consumes gas corresponding to its computational cost. If the total consumption exceeds the prespecified amount, the execution of the program is aborted immediately and the effect of the program execution is rolled back. One who would like to execute a contract is required to pay cryptocurrency equal to the expected amount of the gas consumption in advance, which is not refunded even if the execution is aborted due to an out of gas error. A method to statically analyze the gas consumption is therefore required to avoid such trouble.

Resource Aware ML (RAML) is one of the methods to analyze the execution cost of a program. RAML is a functional programming language equipped with the OCaml syntax. Using the potential-based amortized analysis, which is an analysis of the costs of a sequence of operations whose costs vary depending on the state of the data structure, RAML automatically and statically analyzes the resource consumption bounds for a given program for a specified metric.

This study proposes a method to statically analyze the gas consumption of a smart contract of cryptocurrency Tezos using RAML. A Tezos smart contract is written in a stack-based programming language Michelson. To this end, we implemented a library in RAML each function of which imitates the behavior of each Michelson instruction. Using this library, we can write a RAML program that imitates the behavior of a Michelson program. Then,

we can estimate the gas consumption of the program by analyzing it with RAML.

Since a Michelson instruction rewrites the stack, we implemented a RAML function that imitates each Michelson instruction as one with `(t list` $\rightarrow$ `t list)`, where `t` is the variant type that expresses stack elements and `t list` is the type that expresses a stack. Then we can implement a RAML program that imitates the behavior of a Michelson program as one that applies the functions in our library to the initial stack.

Although the gas consumption of execution of a Tezos contract consists of several types of costs, we aim at estimating the *interpreter cost*, which is for executing a Michelson program. For this purpose, we use the *tick metric* of RAML, which is used for defining the quantity of custom resource consumption. We can define the resource consumption of a function by calling a primitive function provided by RAML. We implemented the library so that each function calls the function that expresses its interpreter cost. By encoding a contract with our library as a RAML program, we can estimate the interpreter cost of the contract by analyzing the RAML program with the tick metric.

We encoded several Michelson contracts to RAML programs using our library and analyzed them to estimate the gas consumption. The analysis was successful for the contracts that consisted only of instructions of basic stack operations and conditional branches; the estimation was correct for these contracts. However, the analysis was not successful for contracts that contain instructions for recursion on lists. RAML could not analyze the interpreter cost of contracts that contain instructions whose gas consumption depends on the contents of the stack.

Extension of the library to cover more Michelson instructions and to handle recursion on lists are left as future work. We also plan to study other types of costs than the interpreter cost and estimate them.

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 **1**

2 背景知識 **2**

2.1 Tezos と Michelson 2

2.2 コントラクトのガス消費の仕組み 7

2.3 Resource Aware ML (RAML) 9

3 RAML での Michelson プログラムの実装 **14**

3.1 文法 15

3.2 命令 16

3.3 ライブラリを用いたプログラム 17

3.4 tick メトリックによるガス消費量の見積もり 19

4 解析例と考察 **21**

5 関連研究 **23**

6 結論 **24**

6.1 結論 24

6.2 今後の課題 24

謝辞 **25**

参考文献 **25**

付録

1 序論

2008年にビットコインが開発されて以来 [1]，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，分散ネットワーク上での取引が活発化している．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 [2]．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．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，ブロックの生成者であるマイナーに対して払われる手数料となる．コントラクトの実行者は，実行前にガス消費量の上限となるガス許容量を設定し，消費量の合計が許容量を超えると，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．また，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．このような無駄なコストを削減し，コントラクトの実行コストを抑えるために，コントラクトの実行者は実際のガス消費量より大きいが，大幅には上回らないガス許容量を設定することが望ましい．そのために，ガスの消費量を静的に解析する手法が求められている．

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を静的に解析する手法を提案する．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，そ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 で記述される．

解析を行う手法として，本研究では *Resource Aware ML (RAML)* [3] を用いる．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手法をもとに設計された [4]，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．

Michelson のプログラムを RAML で解析するために，Michelson の各命令の挙動を模倣するライブラリを RAML で実装する．このライブラリを用いて，Michelson プログラムを模倣する RAML プログラムを作成する．その RAML プログラムを，RAML に備えられているメトリックの一つである tick メトリックを用いて，コントラクトのガス消費量のうち，プログラムの解釈実行を行う際に発生する interpreter cost の見積もりを行う．以上のような手法で，いくつかのコントラクトの解析を行った結果，単純な命令のみで構成されているコントラクトの interpreter cost を正しく見積もることができた．一方，interpreter cost が複雑な計算によって算出されるような命令を含むコントラクトについては見積もりが難しく，今後の課題である．

本報告書は以下のように構成されている．第 2 章では，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 RAML についてそれぞれ説明する．第 3 章では，Michelson プログラムを模倣する RAML プログラムの実装について説明する．第 4 章では，コントラクトのガス消費量の解析についていくつかの例を挙げ，結果と考察を述べる．第 5 章では，本研究の関連研究について論ずる．最後に第 6 章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 2.1 節では Tezos と Michelson について，第 2.2 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 2.3 節では RAML についてそれぞれ述べる．

2.1 Tezos と Michelson

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 [2]．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．

Michelson 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 [5].

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

図 1: Michelson の文法

Michelsonの文法のうち，本研究でRAMLで実装する部分の文法を図1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の通貨量の型`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はスタックの要素の値を表し，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．空のスタックを表す`E`は，要素の積まれたスタックを表すときは省略する．例えば，$V_1 : V_2$は$V_1 : V_2 :$ `E`を省略した形である．

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．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ex-

ample1 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') */
```

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 には静的型検査が備えられている．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付け規則が存在する．例えば，PAIR についての型付け規則は以下のように表される．

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

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 についての型付け規則は以下のように表される．

$$\texttt{int} : \texttt{int} : S' \rightarrow \texttt{int} : S'$$

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付け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検査が行われる．

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関数値，型などの値は全てバイト列として保存され，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 と呼ばれるコストが発生する．
2. バイト列は，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 と呼ばれるが発生する．Micheline 表現では，全ての値が以下の要素で表される．

- integer
- string
- バイト列
- 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 と呼ばれるコストが発生する．
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 と呼ばれるコストが発生する．
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 と呼ばれるコストが発生する．
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 と呼ばれるコストが発生する．
7. Micheline 表現はバイト列へとシリアル化され，保存される．このとき，Ssrialization cost と呼ばれるコストが発生する．
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 と呼ばれるコストが発生する．

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 (RAML)

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上界を，静的かつ自動的に解析する機能をもつ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上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 [4]．計算量の 1 つとして，同じ処理を連続で行ったときの 1 回あたりの計算量を償却計算量という．償却計算量を求めることを償却解析といい，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手法として用いられる．償却解析における手法の一つとして，データ構造に対してポテンシャル Φ と呼ばれる非負の数値を導入する方法がある．ポテンシャルはそのデータ構造に対して貯まっていて，処理に対して支払われる貯金として扱われる．ポテンシャルを導入することで，以下のように償却計算量を求めることができる．

$$\begin{aligned} \text{償却計算量} &= \text{実計算量} + \Delta\Phi \\ &= \text{実計算量} + \Phi(\text{実行後}) - \Phi(\text{実行前}) \end{aligned}$$

以上のようにして行われる償却解析を，ポテンシャルベースの償却解析と呼ぶ．

図 2 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 $\texttt{cons}(x_h, x_t)$，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

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

図 2: RAML の文法

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 [3]，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リソース消費である 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) を呼び出すことができる．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．

```
1  let rec append l1 l2 =
2    match l1 with
3      | [] -> l2
4      | x::xs -> x::(append xs l2)
5
6  let rec partition f l =
7    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

20 行目の `let _ = ...` の部分は main 式を表していて，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，主要な操作として evaluation と analysis の 2 つがある．

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 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，指定した名前の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．

```
$ ./main analyze steps 1 4 -print level 1 examples/quicksort.raml

Resource Aware ML, Version 1.5.0, June 2020

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

Analyzing expression ...

  Trying degree: 1, 2

  Function types:

== quicksort :

  [int -> int -> bool; int list] -> int list

  Non-zero annotations of the argument:
        35  <--  (*, [::(*); ::(*)])
        36  <--  (*, [::(*)])
         3  <--  (*, [])

  Non-zero annotations of result:

  Simplified bound:
     3 + 18.5*M + 17.5*M^2
   where
     M is the number of ::-nodes of the 2nd component of the argument

====

  Derived upper bound: 1624.00

```

```
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，d1 および d2 で指定された次数 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

本章では，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 tick メトリックを用いたガス消費量の見積もりについて述べる．なお，実

装したライブラリは本研究の GitHub レポジトリ [1)] の library/michleson.raml にあるので，適宜参照されたい．

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．

```
1 type t =
2   Int of int | Nat of Rnat.t | Mutez of Rnat.t |
3   Bool of bool | Address | Unit | MNone | MSome of t |
4   LNil | LCons of t * t | Operation | Contract |
5   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数を表す型で，四則演算 add，mult，minus，div_mod と，n が 0 か 1 以上かで分岐する条件分岐関数 ifz が関数として用意されている．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 の値である，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

[1)] https://github.com/yutono326/raml

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，実装した関数のうち，特筆すべきものについて説明する．

- `FAILWITH` は例外 `Invalid_argument` を発生させる関数として定義する．
- `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 メトリックによるによるガス消費量の見積もりにおいて用いる．
- 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．
- `LOOP` は Michelson の挙動を再現すると解析が難しくなるため，以下に示すように挙動を簡単なものにしている．

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

具体的には，本来の引数に加えて `Rnat.t` 型の引数 `n` を受け取って，スタックのトップの要素に関わらず，引数の命令シーケンスを `n` 回，スタックのトップを除いた残りのスタックに適用する関数として定義する．定義中に用いられる `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．

- PUSH は，受け取った要素をスタックに積む命令として定義するが，積む要素の型の指定はなく，値のみを引数として受け取る．
- ADD，SUB，MUL，EDIV は，それぞれスタックのトップと 2 番目の要素を取り出し，整数値の演算結果をスタックに積む関数として定義する．命令にもよるが，スタックのトップと 2 番目の要素の型として，Int 同士，Nat 同士，Int と Nat，Mutez 同士，Mutez と Nat といったパターンがあり，パターンに応じて演算結果の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を行い，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．
- COMPARE は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として定義する．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．
- NONE，LEFT，RIGHT，NIL について，Michelson における命令の定義ではそれぞれオプション値，ユニオン，リストの中身の型を引数で指定する必要があるが，本ライブラリの関数の定義では型指定はしない．
- MAP，SIZE，ITER はリストに対する再帰を行う命令で，本ライブラリにおいてはスタックの要素が命令に合致している場合に，それぞれ対応した補助関数を呼び出す関数として定義し，その補助関数においてリストの再帰を行う．
- TRANSFER_TOKENS，CONTRACT，SOURCE，AMOUNT といったコントラクトに関する命令は，本ライブラリではコントラクトに紐づけられたアドレスや通貨量などの情報を管理することができないので，それらの情報を簡略化したスタック操作のみを行う関数として定義する．

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，初期スタックを表す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.tz の挙動を模倣する RAML プログラム example1.raml を Code4 に示す．

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

Code 4: example1.raml

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ることに注意してほしい．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 library/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: 249.00
  Ticks:            34.00
  Heap space:       143.00
  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 開発コミュニティから提供されている仮想環境[1)]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，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.tz について run script を実行した結果を以下に示す．

```
$ ./tezos-client run script contracts/pairadd.tz on storage 0 and input 'Pair 3 5'
  --trace-stack --gas 100000
storage
  8
emitted operations

trace
  - location: 8 (remaining gas: 99655 units remaining)
    [ (Pair (Pair 3 5) 0)    ]
  - location: 9 (remaining gas: 99652 units remaining)
    [ (Pair 3 5)    @parameter ]
  - location: 10 (remaining gas: 99649 units remaining)
    [ (Pair 3 5)    @parameter
```

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

```
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 `100000 units` と指定していてるが，最初の出力時点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 `3 units` 消費されていることがわかる．ここから `CAR` 命令の `interpreter cost` が `3 units` 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 `1 units` 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 関数を呼び出すよう定義し，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．

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，スタックの要素の整数値の絶対値に依存していて [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 `10000 units` 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，12000.0 の tick 値を発生するように定義されている．

4 解析例と考察

以下の 5 つの Michelson プログラムについて，挙動を模倣する RAML プログラムを，第 3 章で設計したライブラリを用いて実装し，tick メトリックによる解析を行い，interpreter cost の見積もりを行った．

example1.tz Code1 で示したプログラム．

example2.tz `parameter` として受け取った整数値を 0 になるまで 1 ずつ減算して，ストレージに書き込む．

example3.tz `parameter` として受け取った整数値が 0 より大きいなら 1，0 以下なら-1 をストレージに書き込む．

example4.tz `parameter` として受け取った整数値のリストの，要素の和をストレージに書き込む．

example5.tz `source` と呼ばれる取引が始まった実行の起点のアカウントへ受け取った通貨を送金し返す．

Michelson プログラム example2.tz，example3.tz，example4.tz，exmaple5.tz，およびそれぞれを模倣する RAML プログラム example2.raml，example3.raml，example4.raml，exmaple5.raml のコードを付録に付したので，適宜参照されたい．

[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．

解析の結果を，表 1 に示す．interpreter cost は run script コマンドにおいて示された interpreter cost で，derived upper bound は RAML プログラムの解析の結果として得られた上界である．

表 1: interpreter cost の解析結果

プログラム	interpreter cost	derived upper bound
example1.tz	43	37.00
example2.tz	282	225.00
example3.tz	28	28.00
example4.tz	80	failed
example5.tz	12079	12079.00

example1.raml と example2.raml の解析結果については，interpreter cost と derived upper bound に多少の開きがあるが，これは，`ADD` 命令の interpreter cost がスタックの要素の整数値の絶対値に依存している点が RAML の解析において考慮されてないことに起因していて，それを除けば正しく見積もることができたと言える．example4.raml については解析自体が正しく行われなかった．これは，リストに対する再帰を行う命令である `ITER` 命令がプログラムに含まれていたことが原因であると考えられる．example5.raml は `CONTRACT` 命令を含んだプログラムで，正確な解析はできないと先述したが，このプログラムにおける `CONTRACT` 命令の interpreter cost が 12000units であったため，解析結果として正しい上界が得られた．

example4.raml の解析に関して，以下に示すような `iter_list` の関数適用のみを行うプログラムついての解析は行うことができる．

```
let _ =
  iter_list (p |> add |> p) (LCons (Int 1, LCons (Int 2,
    LCons (Int 3, LCons (Int 4, LNil)))) :: Int 0 :: [])
```

このプログラムの解析結果として得られる上界が 32.00 であることから，以下のように example4.raml の `iter_list` の部分を置き換えることで，プログラムの解析を行うことができる．

```
let f s = Raml.tick(32.0); Int 10 :: []
```

```
2
3 let _ =
4   (pair
5   (nil
6   (f
7   (dip1 (p |> push (Int 0) |> p)
8   (car
9   (Pair (LCons (Int 1, LCons (Int 2, LCons (Int 3, LCons (Int
      4, LNil)))), Int 0) :: [])))))))
```

しかし，example4.raml のような `iter_list` を含む関数を連続適用するプログラムについて解析が正しく行われなかった点については，詳細な原因はわからなかった．

5 関連研究

スマートコントラクトを対象とした静的な解析は，本研究で示したガス消費量の解析の他にも様々な目的で研究されている．その中でも，ブロックチェーン技術が幅広い用途に利用されるにつれ，スマートコントラクトが扱うデータの重要性も高まっており，大量の通貨をスマートコントラクト上で取り扱うため，プログラムを静的かつ形式的に検証する手法への需要が高まっている．

西田らは，Tezos のスマートコントラクト言語 Michelson の静的型検証ツール Helmholtz を開発した [6] [7]．Helmholtz の開発にあたって，Mihchelson のための篩型システムを単純型システムの拡張として実装している．この篩型システムは，スタックの型に対して条件を付けるシステムである．Helmholtz は，篩型の形式で表現され，ユーザーの定義した注釈が付けられた Michelson プログラムを受け取る．次に，篩型システムに基づく仕様の注釈の入ったプログラムに対する型検査が行われ，型検査中に生成された検証条件は SMT ソルバである Z3 [8] によって検証される．Helmholtz の検証において，コードが正しく型付けされているプログラムは仕様を満たすことが保証される．

6 結論

6.1 結論

本研究では，RAML を用いて，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した．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である Michelson で記述されていて，その挙動を RAML で再現するために，スタックの要素を独自のヴァリアント型の値として定義し，MIchelson の各命令を，スタックを表すリストを受け取ってその内容を変更して返す関数として定義し，Michelson プログラムを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 RAML プログラムとして実装した．ガス消費量の見積もりについては，コントラクトの実行に伴うガス消費量のうち interpreter cost を見積もることを目的とし，RAML の tick メトリックを用いてプログラムの解析を行う手法を示した．

いくつかのコントラクトについて解析を行い，interpreter cost を見積もった結果，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，ガス消費量がスタック中の値に依存す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．

6.2 今後の課題

本研究では，Michelson に実装されている命令のうち主要な命令について RAML ライブラリで扱えるように実装した一方で，set や map，関数などの型は実装しておらず，それらに関連する命令も実装されていない．本ライブラリにおいてより多くのプログラムを扱えるように，実装を進めていく予定である．また，リストの再帰を行う命令を含むコントラクトの解析が正しく行えなかった点について，リスト型の実装方法に問題があった可能性がある．これを正しく解析できるようにするために，実装体系を見直すことも検討している．

また，コントラクトの実行に伴うガス消費量のうち，interpreter cost を見積もる手法を示したが，interpreter cost はコントラクトの実行コスト全体のごく一部に過ぎない．コントラクトの実行コスト全体の見積もりを行うために，そ

の他のコストの計算方法について研究し，解析を行うことも今後の課題である．

謝辞

本報告書の執筆にあたり，多くの方々に支援いただきました．本研究に取り組むにあたって，研究テーマを提示して頂き，終始適切な助言を賜り，丁寧に指導して下さった末永幸平准教授に心より感謝いたします．また，Tezos や Michelson の体系や様々な知識について解説して頂いた古瀬淳氏と，基本的な Michelson コードを提供して頂いた斉藤大聖氏に感謝いたします．そして，本報告書の執筆にあたって，添削や助言を頂いた五十嵐淳教授，和賀正樹助教をはじめ，多くの助言を頂いた五十嵐・末永研究室の皆様に深く感謝いたします．

参考文献

[1] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system, `https://bitcoin.org/bitcoin.pdf` (2008). (Accessed on January 29th, 2021).

[2] Allombert, V., Bourgoin, M. and Tesson, J.: Introduction to the Tezos Blockchain, *17th International Conference on High Performance Computing & Simulation, HPCS 2019, Dublin, Ireland, July 15-19, 2019*, IEEE, pp. 1–10 (2019).

[3] Hoffmann, J., Aehlig, K. and Hofmann, M.: Resource Aware ML, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings* (Madhusudan, P. and Seshia, S. A.(eds.)), Lecture Notes in Computer Science, Vol. 7358, Springer, pp. 781–786 (2012).

[4] Hoffmann, J. and Hofmann, M.: Amortized Resource Analysis with Polynomial Potential, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (Gordon, A. D.(ed.)), Lecture Notes in Computer Science, Vol. 6012, Springer, pp. 287–306 (2010).

[5] Michelson: the language of Smart Contracts in Tezos,

https://tezos.gitlab.io/008/michelson.html. (Accessed on January 29th, 2021).

[6] Nishida, Y., Saito, H., Chen, R., Kawata, A., Furuse, J., Suenaga, K. and Igarashi, A.: Helmholtz: A Verifier for Tezos Smart Contracts Based on Refinement Types, *27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021)* (2021). To appear.

[7] 齋藤大聖: 篩型を用いた Tezos スマートコントラクト検証器, 修士論文, 京都大学大学院情報学研究科通信情報システム専攻 (2021).

[8] de Moura, L. M. and Bjørner, N.: Z3: An Efficient SMT Solver, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (Ramakrishnan, C. R. and Rehof, J.(eds.)), Lecture Notes in Computer Science, Vol. 4963, Springer, pp. 337–340 (2008).

付録

第 4 章において用いた Michelson プログラムと，それを模倣する RAML プログラムのコードを以下に示す.

```
parameter int;
storage int;
code {  CAR ;
        DUP ;
        GT ;
        LOOP { PUSH int -1 ; ADD ; DUP ; GT } ;
        NIL operation ;
        PAIR  }
```

Code 5: example2.tz

```
parameter int;
storage int;
code {  CAR ;
        GT ;
        IF {PUSH int 1} {PUSH int -1} ;
        NIL operation ;
        PAIR  }
```

Code 6: example3.tz

```
parameter (list int);
storage int;
code {  CAR ;
        DIP { PUSH int 0 } ;
        ITER { ADD } ;
        NIL operation ;
        PAIR  }
```

Code 7: example4.tz

```
parameter unit;
storage unit;
code {  CDR ;
        NIL operation ;
        AMOUNT ;
```

```
        PUSH mutez 0 ;
        COMPARE ;
        EQ ;
        IF
          {}
          { SOURCE ; CONTRACT unit ; { IF_NONE FAILWITH {} }
            ; AMOUNT ; UNIT ; TRANSFER_TOKENS ; CONS } ;
        PAIR  }
```

Code 8: example5.tz

```
let _ =
  (pair
  (nil
  (loop (Rnat.of_int 10) (p |> push (Int (-1)) |> add |> dup
    |> gt |> p)
  (gt
  (dup
  (car
  (Pair (Int 10, Int 0) :: [])))))))
```

Code 9: example2.raml

```
let _ =
  (pair
  (nil
  (if_ (p |> push (Int 1) |> p) (p |> push (Int (-1)) |> p)
  (gt
  (car
  (Pair (Int 5, Int 0) :: [])))))))
```

Code 10: example3.raml

```
let _ =
  (pair
  (nil
  (iter_list (p |> add |> p)
  (dip1 (p |> push (Int 0) |> p)
  (car
  (Pair (LCons (Int 1, LCons (Int 2, LCons (Int 3, LCons (Int
    4, LNil)))), Int 0) :: [])))))))
```

Code 11: example4.raml

```
let _ =
  (pair
  (if_ (p |> p) (p |> source |> contract |> if_none failwith
   (p |> p) |> amount |> unit |> transfer_tokens |> cons |>
   p)
  (eq
  (compare
  (push (Mutez Rnat.zero)
  (amount
  (nil
  (cdr
  (Pair (Unit, Unit) :: [])))))))))
```

Code 12: example5.raml