特別研究報告書

スマートコントラクトのガス消費量の Reource 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 Reource 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変更が取り消される．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．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．

解析の手法の 1 つとして，ポテンシャルベースの償却解析がある．償却解析は，コストがデータ構造の状態に依存するような，連続して行われる操作の平均コストを求める手法である．ポテンシャルベースの償却解析は，データ構造に対してポテンシャルという値を対応づけて行う償却解析である．Resource Aware ML(RAML) は，ポテンシャルベースの償却解析を用いて設計された，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範囲（bound の訳を範囲としているが，しっくりこない）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解析することができる．

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の命令は初期スタックを受け取ってスタックの内容を変更して返す関数として実装されている．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．このコントラクト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した．

このプログラムに対して，リソース消費量の解析を行った結果，基本的な命令のみで構成されているコントラクトについては正しく解析が行われたが，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．コントラクトのガス消費量の見積もりについては，RAMLのtickメトリックを用いた．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，その命令のガス消費量に相当する値の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 costsを見積もることに成功した．

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，ガス消費量の見積もりについてはinterpreter costについてのみ取り組んだが，コントラクトの実行コストに含まれ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

スマートコントラクトのガス消費量の Reource 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．なお，スタックの状態を記述する際，スタックの要素を : で区切って表す．

- 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，以下の 8 つの段階でガスの消費が発生する．

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．
5. 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．
6. 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．
7. 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

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

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

$$
\begin{aligned}
\mathrm{A} \;&::=\; \mathrm{unit} \mid \mathrm{bool} \mid \mathrm{int} \mid \mathrm{L(A)} \mid \mathrm{(A, A)} \\
\mathrm{F} \;&::=\; \mathrm{(A, ..., A)} \to \mathrm{A}
\end{aligned}
$$

図 1: 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，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 関数が呼び出される度に，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

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．

```
$ ./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:
```

```
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．

3　RAMLでのMichelsonプログラムの実装

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, 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).