Vol.2020-DPS-182 No.8
Vol.2020-CSEC-88 No.8
2020/3/12

ZoKratesのドメイン固有言語に対するLispベースの拡張とVariant型の提案 (2020年02月19日版)

岡 大貴　高野 祐輝　鄭 振牟　宮地 充子

概要：分散型の合意形成システムであるブロックチェーンは当初の用途であるビットコインにとどまらず，ブロックチェーン上にプログラムを記述できるイーサリアムなど様々な領域に拡張されている．原理的にブロックチェーン上の情報はすべて外部に対して公開されている．そのため，外部に公開できない情報をブロックチェーン上で扱うことができないという課題がある．機微な情報を扱う技術としてゼロ知識証明の技術を用いた手法が研究されている．ZoKrates と呼ばれる統合ツールボックスでは，イーサリアム上のスマートコントラクトをオフチェーンで記述・実行し，その結果が正しいことをゼロ知識証明を用いてブロックチェーン上で検証するモデルが提案されている．ZoKrates ではドメイン固有言語 (DSL) が提供されており，ユーザは証明を生成するための複雑な技術を把握することなくプログラムを記述できる．しかしながら，実装されている言語は基礎的なものである．本研究の目的は ZoKrates の DSL を拡張をすることでツールの機能性を向上させることである．実際に Lisp をベースとした拡張言語を実装し，拡張機能の一例として Variant 型を提案する．Variant 型を用いることでより正確な意味付けが可能となり，実行時エラーを低減することができる．

Lisp-based Extension to Domain-specific Language in ZoKrates and Proposal of Variant Type (version 2020/02/19)

Abstract: As a decentralized consensus system, blockchain is not limited to its original use of Bitcoin, but has been extended to various areas such as Ethereum, on which we can write programs as smart contract. In principle, all information on the blockchain is open to the public. Therefore, there is a problem that private information which cannot be disclosed to the public cannot be handled. For handling private information, a technique using a zero-knowledge proofs has been studied. In an integrated toolbox called ZoKrates, a model has been proposed in which smart contracts on Ethereum are executed off-chain and the results are verified on the blockchain using zero-knowledge proofs. ZoKrates provides a domain-specific language (DSL), which allows users to write programs without having to know difficult-to-use low level abstractions to generate proofs. However, ZoKrates DSL is primitive and have room for extension. The purpose of this research is to improve the functionality of the ZoKrates by extending DSL. We actually implements a Lisp-based extension language and it supports the Variant type as an extended function. By using the Variant type, we can write programs more semantically and reduce runtime errors.

1. 研究背景

ブロックチェーンは信頼できる中央集権に依存することなく合意形成が可能なシステムとして 2008 年に提案された．仮想通貨と呼ばれるビットコイン [1] はその最初の用途であり，当初ブロックチェーンの提案はビットコインの提案であった．ビットコインの移送は主に送信ユーザと受信ユーザを示す署名と公開アドレス，どれくらいの量が移動したかという情報によって構成されるトランザクションによって記述される．ブロックチェーンはこのようなトランザクションがひとまとまりに記述された台帳であり，

ⓒ 2020 Information Processing Society of Japan

分散台帳技術とも表現される．第二世代のブロックチェーンと呼ばれるイーサリアム [2] ではビットコインにおけるユーザと同等の概念としてコントラクトと呼ばれるプログラムを記述できるようになった．ユーザは他のユーザに対してビットコインを送信するように，ブロックチェーン上のコントラクトに対してトランザクションを送信することが可能となった．コントラクトに対するトランザクションには仮想通貨だけでなく，プログラムの入力としてのデータを追加することができる．これによりイーサリアム上では中央集権を介さない様々な自動処理が可能となり，数多くの分散型アプリケーションが開発されている．これらの概念はスマートコントラクトと呼ばれる．

原理的にブロックチェーン上の情報はすべて外部に対して公開されている．そのため，外部に公開できない情報をコントラクトに対する入力として用いることができないという課題がある．ブロックチェーン上で機微な情報を扱う技術としてゼロ知識証明の技術を用いた手法が研究されている．ZoKrates[3] と呼ばれる統合ツールボックスでは，イーサリアム上のスマートコントラクトをオフチェーンで記述・実行し，その結果が正しいことをゼロ知識証明を用いてブロックチェーン上で検証するモデルが提案されている．ZoKrates ではドメイン固有言語 (DSL) が提供されており，ゼロ知識証明の証明を生成するための複雑な技術をユーザが把握する必要なくプログラムを記述することができる．

しかしながら，提供されている DSL は一般的な高級言語に比べると基礎的なものである．データ型は整数型として考えてよい field 型と bool 型，それらの構造体や配列のみであり，浮動小数点や文字型，ポインタ型などは提供されていない．また、スタックやキュー，二分探索木やハッシュテーブル，ヒープといったデータ構造も提供されていない．関数の記述や，コントロールフローとして if 文と for 文がサポートされているが，その用法は限られている．オブジェクト指向言語におけるクラスの定義や，そのメソッドでデータを処理していくような機能は提供されていない．プログラムに対する入力はプログラム実行時にコマンドラインから渡す場合に限られている．このように ZoKrates の DSL には拡張の余地がある．

2. 本研究の目的

本研究の目的は ZoKrates の DSL を拡張してより多くの機能をサポートし，ツールの機能性を向上させることである．Lisp[4] をベースとした拡張言語を提案し，この拡張言語から DSL を生成するコンパイラを実装する．また，拡張機能として Variant 型 [5] のサポートを提案する．Variant 型を用いることでより正確な意味付けができるようになり，Variant 型が提供されていない言語で実行時エラーとなりうるプログラムをコンパイル時の型検査で発見できるという利点がある．

3. 本論文の構成

最後に本論文の構成について記載する．第 4 章では Lisp のインタープリタや Variant 型に関する基礎知識について述べる. 第 5 章ではゼロ知識証明の技術をブロックチェーン技術と組み合わせたオフチェーン実行モデルを提供することで機微なデータを秘匿することができる統合ツールボックスである ZoKrates とその DSL の仕様について述べる．第 6 章では本研究が提案する Lisp をベースとした拡張言語と Variant 型の仕様について述べる．第 7 章では Lisp をベースとした拡張言語における Variant 型を ZoKrates の DSL で実装する方法と Lisp をベースとした拡張言語を DSL に変換するコンパイラの構築について述べる．第 8 章でまとめと今後の課題を述べる．

4. 準備

4.1 Lisp

Lisp は 1958 年に設計されたプログラミング言語であり，動的型付けや再帰関数，セルフホスティングやコンパイラなど，計算機科学において重要な概念を多く開拓した。名前は List Processor に由来している。リストは Lisp の主要なデータ構造であり，また Lisp のソースコード自体もリストで構成されているため，ソースコードをそのままデータとして扱うことができるという特徴がある。このデータとソースコードを同様に扱えるという利点から，マクロの記述や Lisp を対象としたコンパイラ構築における入出力や抽象構文木 (AST) の解析が他の言語に比べて平易である。また，演算は前置記法（ポーランド記法）で記述され，多くの場合においてリストの最初の要素が演算子や関数名，特殊形式になっていることもコンパイラ構築が平易な理由である。初期の Lisp から現代に至るまで大きく変化しており，Scheme や Common Lisp といった様々な方言が派生している。

4.2 Mal - Make a Lisp

Mal[6] とは make a lisp の頭文字であり，様々な言語で Lisp のインタープリタを作成するプロジェクトであり，同時にこのプロジェクトで作成する Lisp 系言語の名前である．約 80 の言語でインタープリタの実装が公開されている．Mal の実装は Lisp の中心となる機能を表した 11 のステップに分割されており，最後のステップでセルフホスティングが可能となる.

4.2.1 AST の表現

Mal では入力に対して以下のような AST を生成する．仕様では define は def!，lambda は fn*と表現されるが，ここではより一般的なキーワードを用いて記述した．第 4.1 節で記述したように Lisp における AST の表現はソースコードをそのまま表現したものだと言える．Lisp のソースコードにおける ( ) に対応するものが List になり，AST のノードとなる．それ以外の要素はすべて木構造の葉となる．

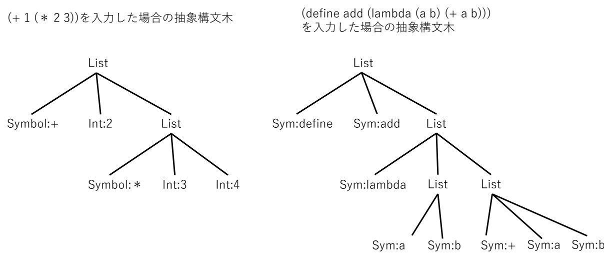

図 1 Mal における抽象構文木の例

4.3 直和型 (Variant 型)

Variant 型は格納する可能性のある型を複数指定でき，一度にそのうちの一つの型の値を格納できる．タグ付き共用体や Variant 型とも表現される．Variant 型は以下のように記述される．

$$Type1 \mid Type2 \mid Type3 \mid ...$$

4.3.1 Variant 型によるパターンマッチング

Variant 型が有効な例としてファイル入出力の例を示す．図 2 は指定した名前のファイルが存在すればそのファイルに書き込み，存在しなければエラーを出力するコードである．

```
int main() {
    char buf[512] = {};
    int fd = open("foo.txt");

    // ファイルが存在する場合、書き込む
    if (fd >= 0) {
      write(fd, buf, sizeof(buf));
    } else {
    // ファイルが存在しない場合、エラーを出す
      printf("error");
    }
    return 0;
}
```

図 2 ファイルを読み込み，書き込む正常なコード

int 型のファイル識別子である fd はファイルが存在しないときに負の値を返している．fd の値で条件分岐をすることで正しいエラー処理が行われている．次に図 3 にエラー処理のないコードを示す．

```
int main() {
    char buf[512] = {};
    int fd = open("foo.txt");

    // 条件分岐を行わず，書き込みを行う．
    write(fd, buf, sizeof(buf));
    return 0;
}
```

図 3 ファイルを読み込み，書き込む実装ミスのあるコード

このようにエラー処理を書き忘れてしまった場合，実行時エラーが起こりうる．指定したファイルが存在しているかどうかが実行時にしか判明せず，コンパイル時にそれを検査することができないためである．この例で示したような単純なコードであればコンパイル時のエラーと実行時のエラーでそれほど違いがないかもしれない．しかしながら，実行に時間のかかるプログラムや，このコード部分の実行頻度が少ないプログラムにおいては大きな問題になる．

これは open 関数が int 型を返す関数として実装されており，fd が int 型の値しか持つことができないことに由来する問題である．open 関数が Variant 型を返す関数として実装され，ファイルが存在していた場合に正の値を返し，存在しなかった場合に void 型を返す関数として設計されている場合を考える．このような実装とパターンマッチングが提供されている場合，上記のような実装ミスをコンパイル時に発見することができる．図 4 に Variant 型が提供されている場合の擬似コードを示す．

```
int main() {
    char buf[512] = {};
    // intもしくはvoid型をもつVariant型の宣言
    // open関数はファイルが存在するときに
    // 正の値を返し，存在しない場合void型を返す
    int | void fd = open("foo.txt");

    // Variant型のパターンマッチング
    match(fd){
      //fdがint型の場合，int型として取り出す
      int: write(fd, buf, sizeof(buf));
      void: printf("error");
    }
    return 0;
}
```

図 4 Variant 型を用いた，ファイルを読み込み，書き込む擬似コード

match 文は Varinat 型である fd の保存している値の型に対応した型で値を取り出すことができる．match 文を記述しなかった場合，write 関数の引数となる fd が Variant

型のままなので int 型を引数とする write 関数はコンパイル時にエラーを出力する．この場合を図 5 に示す．

```
int main() {
    char buf[512] = {};
    // intもしくはvoid型をもつVariant型の宣言
    int | void fd = open("foo.txt");

    // エラー処理を書き忘れた場合
    // コンパイル時にエラーが起こる
    // int型を第一引数に取るwrite関数は
    // Variant型のfdが与えられ
    // コンパイルエラーを起こす
    write(fd, buf, sizeof(buf));
    return 0;
}
```

図 5 Variant 型を用いてエラー処理を書かなかった場合のコード

以上の例のように，Variant 型は実行時にならないと入力が定まらず，入力に応じて意味の異なる返り値を必要とする関数をより正確に設計することができる．これはより正確な意味付けができることを示しており，Variant 型が提供されていない言語では実行時エラーとなりうるプログラムをコンパイル時の型検査で発見できるという利点がある．

5. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations

本章ではゼロ知識証明を用いてスマートコントラクトのオフチェーン実行モデルを導入することで，Ethereum のトランザクションのスループットを向上させ，データを秘匿することができる統合ツールボックスである ZoKrates について述べる．

5.1 概要

スケーラビリティと機密性はイーサリアムの課題であり，要因と課題をまとめると以下である．

- トランザクションがすべてのノードで実行されるためにシステムの規模が大きくなりにくい
- イーサリアム上の情報がすべてのノードに公開されているため，機密性の高いデータを扱うことができない

この論文では以下の二つの貢献をしている．

(1) トランザクションスループットの向上と秘匿性を提供するオフチェーンの計算モデルの提案

(2) 上記の計算モデルを統合したツールボックスである ZoKrates の実装

1 では非対話型のゼロ知識証明を用いてオンチェーンのスマートコントラクトをオフチェーンで実行し，オンチェーンではその結果の検証のみを実施するモデルを提案する．すべてのノードで実行されていたスマートコントラクトはオフチェーンで実行され，証明の検証のみオンチェーンのすべてのノードで行われるようになる．よって検証のスマートコントラクトが元のスマートコントラクトよりも計算量が少ない場合にトランザクションのスループットが向上する．またゼロ知識証明の性質により，オフチェーンで行われた計算の入力等の情報は外部に開示されないため，秘匿性を獲得する．2 ではオフチェーンの計算モデルを統合したツールボックスである ZoKrates を提案する．ZoKrates は DSL，コンパイラ，証明の生成，検証用のスマートコントラクトの生成から構成されている．DSL が提供されることにより，ユーザーはゼロ知識証明の証明を生成するための高度な抽象化を把握する必要がない．

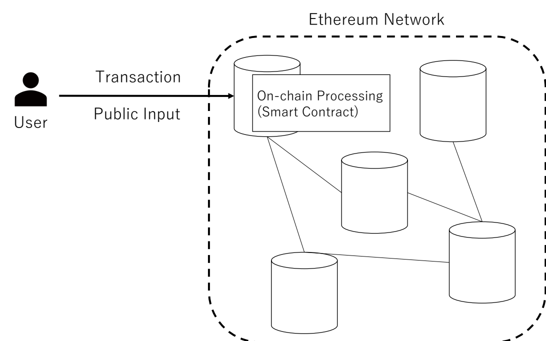

図 6 従来の on-chain processing

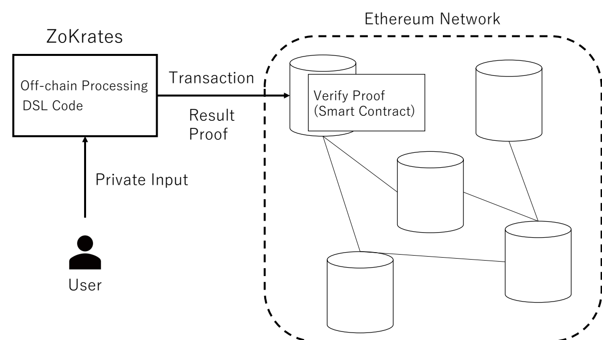

図 7 ZoKrates を用いた off-chain processing

5.2 ZoKrates のドメイン固有言語

(1) 構造

main 関数から実行される．この関数は private と public な入力をもっており，少なくとも一つの値を返す．public な入力は証明の検証の際にブロックチェーンに送信されて公開されるが，private な入力は外部に公開されることがなく秘匿性の高いものを扱うことができる．任意の関数が定義でき，呼び出しすることができる．関数は定義が呼び出しより先にある必要があ

り，静的なスコープを持つ．

(2) Types

ZoKrates の基本的なデータ型はある素数に対する剰余体の要素である field 型である．これはある固定の素数の剰余で表される 0 以上の整数である．254 ビットの素数が使用されており，開発者は実質的に符号なし整数としてコードを記述することができる．boolean 型は剰余体の 0 と 1 のみを取り，true と false で表される．そのほかに固定長の配列や，構造体が利用することができる．

(3) 演算子

基本的な四則演算と等価演算子が実装されている．

(4) コントロールフロー

For ループと if-else 文が実装されている．if-else 文のそれぞれの節には値として評価されるものしか記述できないため，代入が主な使い方となる．また，節の中に複数行記述することができない．For ループについてはコンパイル時にループが展開されるため，ループ回数の上限値が定められている必要がある．

(5) その他

ハッシュ関数や公開鍵暗号に使用する関数が標準ライブラリとして提供されている．global 変数は実装されていない．入出力はコンパイルしたコードをコマンドラインで実行するときに指定する方法のみで，ファイル入出力等は実装されていない．

6. DSL に対する Lisp をベースとした拡張

本章では既存の DSL に対する Lisp をベースにした拡張言語と，Variant 型の提案について述べる．第 1 章や第 5.2 節で記述したように，ZoKrates の DSL では基礎的な機能しか提供されていない．DSL 自体に新たな構文を定義する機能はないため、機能を追加するためには新たな拡張言語の仕様を定め，その言語で記述した構文を DSL で表現可能な形にコンパイルする必要がある．本研究では Lisp をベースとした拡張言語を提案する．第 4.1 節に記述したように Lisp はコンパイラ構築における AST の生成とその評価をする関数の実装が他の言語に比べて平易である．本研究では ZoKrates の DSL 拡張における最初の段階として Lisp をベースとした拡張言語を採用した．

6.1 コンパイラの設計

コンパイラの構築には第 4.2 節で記述した Mal や An incremental approach to compiler construction[7] を活用できる．Mal は Lisp インタープリタであり，ソースコードの入力から字句解析，抽象構文木 (AST) の構築とその解析というコンパイラと共通したステップを学ぶことができる．An incremental approach to compiler construction では整数のコンパイルという最小のソースコードから一段階ずつコンパイル対象となる Scheme ソースコードを拡張していくという手法を学ぶことができ，コンパイラ構築全般に活用することができる．

本研究ではまず，Lisp インタープリタである Mal を実装したのち，このインタープリタを ZoKrates DSL に対するコンパイラに拡張するという方針をとる．まず，Mal の step4_if_fn_do までを JavaScript で記述し，ローカル変数の定義や関数の定義，条件式など基本的な文法を実行できるインタープリタを実装する．次にこのインタープリタにおいて構文解析した結果を JavaScript で実行している部分を ZoKrates の DSL 出力に置き換えていく．図 8 に Lisp ソースコードを DSL のソースコードにコンパイルする一例を示す．

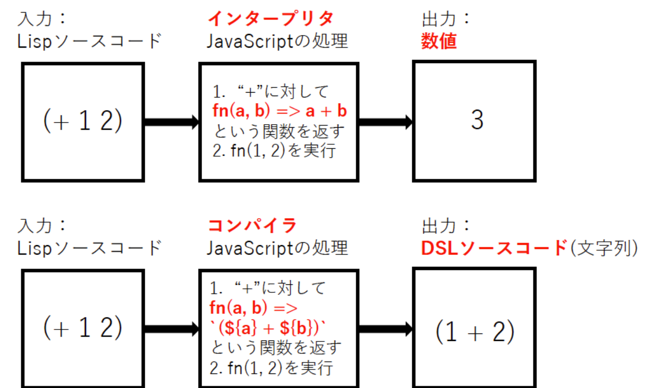

図 8　インタープリタからコンパイラへの拡張

6.2 Lisp ベースの拡張言語の仕様

本章では Lisp をベースとした言語の仕様を説明する．すべての仕様について記載することはできないので，Variant 型に関わる部分のみ記載する．拡張言語において Variant 型の変数は図 9 のように宣言する．また，パターンマッチングは図 10 のような match 文を用いる．

```
(let ((| Type1 Type2) 変数名 初期値)
  (
    (式)
  )
)
```

図 9　拡張言語における Variant 型の宣言

7. ZoKrates における Variant 型の実装

4 章で設計した Lisp をベースとした言語で Variant 型を用いて記述した内容を ZoKrates DSL で表現すると以下のようになる．Variant 型の宣言は構造体を用いて実装

ⓒ 2020 Information Processing Society of Japan

```
(match (Variant型の変数)
  (Type1 (式))
  (Type2 (式))
  ...
)
```

図 10　拡張言語における Variant 型のパターンマッチング

した．この構造体は二つのメンバを持ち，一つ目メンバである type は現在保存しているデータの型の種類を示し，二つ目のメンバは保存しているデータの値を示す．Lisp ベースのプログラムにおいて，match 構文を用いて記述した Variant 型のパターンパッチングについては struct の第一要素である type を評価して場合分けをする if 文として表現した．

```
// Variant型に対応する構造体の宣言
// type:0 -> field
// type:1 -> void
struct variant {
    field type
    field value
}

def main() -> ():
  variant v = variant {type:0, value:0}
  variant v = variantを返す関数()
  // 拡張言語におけるVariant型のmatch文を
  // if式に変換して出力する
  return if v.type == 0 then (式1) else (式2) fi
```

図 11　ZoKrates の DSL における Variant 型の実装

また拡張言語における Variant 型を ZoKrates の DSL で表現した具体例を以下に示す．以下のソースコードは配列の中にある値があるか走査し，値が存在すればそのインデックスを field 型で返し，値が存在しなければ void 型を返す例である．図 12 は ZoKrates の DSL において配列の走査で実行時エラーが起こる例であり，図 13 と図 14 にはそれぞれ，拡張言語において Variant 型を用いて配列を走査するソースコードと，拡張言語における Variant 型を ZoKrates の DSL で表現したソースコードを示した．

8. まとめ

ZoKrates と呼ばれる統合ツールボックスでは，イーサリアム上のスマートコントラクトをオフチェーンで記述・実行し，その結果が正しいことをゼロ知識証明を用いてブロックチェーン上で検証することができる．ZoKrates ではドメイン固有言語 (DSL) が提供されているが，この言語は基本的な機能しかサポートしていない．本研究ではこの DSL に対して Lisp をベースとした拡張言語を提案し、実装した．また拡張機能の例として Variant 型を提案した．Variant 型を用いることで複数の型を返す可能性のある関数の記述ができるようになり，より正確な意味付けが可能となった．また，実行時エラーが発生してデバックに時間がかかる可能性を低減した．

```
// field型を返す関数
def search(field val, field[10] A) ->
  (field):
    // valがないことを示す初期値
    field res = 11
    bool found = false
    for field i in 0..10 do
        found = if !found && A[i] == val
          then true else false fi
        res = if found then i else res fi
    endfor
    return res

def main(field val, field[10] A) ->
  (field):
  field index = search()
  // valが存在せず,
  // indexが11の場合
  // 実行時エラー
  return calc(A[index]);
```

図 12　ZoKrates の DSL において配列の走査で実行時エラーが起こる例

本研究では DSL における構造体を用いて Variant 型を実装した．ある機能を拡張する際，その機能が DSL でどのように実装できるのか考える必要がある．ZoKrates の DSL に対する理想的な拡張機能と実装可能性を考察していくことが今後の課題である．

9. 謝辞

本研究の一部は科学技術振興機構（JST）の CREST(JPMJCR1404) 及び文部科学省「Society5.0 に対応した高度技術人材育成事業成長分野を支える情報技術人材の育成拠点の形成 (enPiT)」さらに文部科学省の平成 30 年度「Society 5.0 実現化研究拠点支援事業」の助成を受けています．

参考文献

[1] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.
[2] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-White-Paper, 2014.
[3] Jacob Eberhardt and Stefan Tai. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. *2018 IEEE International Conference on Internet of Things*

```
; Variant型を返す関数
(define ((| field void)) search
  (lambda (field val field A[10])
    (let (bool found false field res 0)
      (for (field i 0 10)
        (if (= val A[i]))
          (set res i)
        )
      )
      (if found
        res
        nil
      )
    )
  )
)


(define main
  (lambda (field val field A[10])
    ; Variant型の宣言
    (let ((| field void) index (search
      val A))
      ; 以下のパターンマッチを
      ; 書き忘れた場合でも
      ; コンパイル時の型検査で判明する
      ; (match (index)
      ;   (field (calc A[index]))
      ;   (void )
      ; )
      (calc A[index])
    )
  )
)
```

図 13　拡張言語において Variant 型を用いて配列を走査する例

```
// Variant型に対応する構造体の宣言
// type:0 -> void
// type:1 -> field
struct variant {
    field type
    field value
}

def search(field val, field[10] A) ->
  (variant):
    variant res =  variant {type: 0,
      value: 0}
    bool found = false
    for field i in 0..10 do
      found = if !found && A[i] == val
        then true else found
      res = if found then variant {type:
        1, value: i} else res fi
    endfor
    return res

def main(field val, field[10] A) ->
  (field):
    variant index = {type:0, value:0}
    index = search(val, A)
    return if index.type == 1 then
      index.value else 0 fi
```

図 14　拡張言語における Variant 型を ZoKrates の DSL で表現した例

(iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 1084–1091, 2018.
[4] John McCarthy. A Micro-Manual for LISP - Not the Whole Truth. *SIGPLAN Not.*, Vol. 13, No. 8, p. 215–216, August 1978.
[5] Benjamin C. Pierce. 型システム入門 プログラミング言語と型の推論. オーム社, 2013.
[6] mal - Make a Lisp. https://github.com/kanaka/mal.
[7] Abdulaziz Ghuloum. An Incremental Approach to Compiler Construction. 01 2006.
[8] Mastering Bitcoin 2nd Edition - Programming the Open Blockchain. https://github.com/bitcoinbook/bitcoinbook.
[9] Report on Algorithmic Language Scheme: R5RS. https://schemers.org/Documents/Standards/R5RS/.

ⓒ 2020 Information Processing Society of Japan