軽量形式手法 VDM によるバーチャルマシンの開発

小田 朋宏
株式会社 SRA
tomohiro@sra.co.jp

荒木 啓二郎
熊本高等専門学校
araki@kyudai.jp

要旨

バーチャルマシン *(VM)* はソフトウェアで実装された仮想コンピュータシステムであり，プログラミング言語の実行系アーキテクチャとして広く採用されている．*VM* はユーザプログラムを実行するための基盤であり，高い信頼性と処理速度の両立が求められることから，その開発には高度に専門化された高い技能が要求されるとともに，安定した実装が得られるまで数年から *10* 年以上の長い期間を要することが多い．本稿では，*VM* の仕様記述に実行可能な形式仕様記述言語 *VDM-SL* を適用した事例として，現在進行している *ViennaVM* の開発を紹介し，*VM* 開発における軽量形式手法の効果を議論する．

1. はじめに

バーチャルマシン (以下，VM) はプログラミング言語の処理系の実装で採用されている実行系アーキテクチャである [1]．VM は仮想的なコンピュータシステムであり，命令セットやメモリモデル，外部リソース，外部入出力などを持つ．VM の命令セットはバイトコードまたは中間コード (以下，IR コード) セットと呼ばれ，メモリモデル，外部リソース，外部入出力と併せて，しばしば特定のプログラミング言語（以下，ゲスト言語）向けに設計される．例えば，Java VM (以下，JVM) はゲスト言語として Java 言語を想定して設計された IR コードセットを持つ [2]．JVM の IR コードセットは Java 言語でのメソッドの種類および呼び出し方法に応じて，invokedynamic，invokeinterface，invokespecial，invokestatic，invokevirtual など，それぞれ個別に特化されたメソッド呼び出しのための IR 命令が定義されている．プログラミング言語に特化しつつコンパクトな IR コードセットを定義することで，プラットフォーム間の移植性の高さと，階層化されたメモリを持つ現代的な計算機アーキテクチャにおける効率的な実行を両立した処理系を実装することができる．同一ソースプログラムを複数プラットフォーム上で動作させる場合，VM を界面として，プラットフォーム依存な部分を VM に集中させ，VM 上で動作する IR コード列はプラットフォーム非依存にすることができる．同一仕様の VM を複数プラットフォーム上に実装して，プログラムソースから単一のコンパイラを利用して IR コード列を生成し，IR コード列を複数プラットフォーム上で動作させることができる．

VM には利点がある一方で，その開発には高い技量を要するという課題がある．VM の仕様，特に IR コードセットの定義およびレジスタとメモリモデルの設計は，ゲスト言語を効率的に実行する上で重要である．シンプルかつゲスト言語の実行モデルに適合した IR コードセットを定義することで，ゲスト言語の単純なインタプリタよりもコンパクトな処理系を実現することができ，かつ，そのコンパクトさによって階層化されたメモリを持つ現代的な計算機アーキテクチャでの実行効率の改善が期待できる．レジスタおよびメモリモデルについても，高性能な VM を実現するためには，適切な抽象度により IR コードセットを単純化し，かつ，ゲスト言語でのメモリ使用の特性に適合する，オーバーヘッドの少ないシンプルなメモリモデルを設計する必要がある．

VM の仕様には高い厳密性が求められる．VM の利点の 1 つである移植性の高さは，同一仕様の VM はいずれの実装でも同一の仕様に従って動作することに依拠している．VM の仕様に解釈上の曖昧さがあると，ゲスト言語レベルにおいて，実装ごとに異なった振る舞いをする可能性がある．

実行系としての VM には処理速度と信頼性の高いレベ

ルでの両立が求められる．VM の実装には効率的な実行のために技芸的な技術がしばしば適用され，仕様に対して正しい実装となっているかを確認することは容易ではない．個々の IR コードの実装の正しさを確保するためには，適切な粒度と十分な厳密さを持つ IR コードセットの定義が求められる．メモリ管理，外部リソースや外部 IO についても同様である．また，IR コードレベルの最適化や IR コードからネイティブコードへの JIT コンパイルなどの技術が普及しつつあるが，これらの技術も VM 仕様に依拠したものである．

ViennaVM[4] は著者らが開発中の，実行可能形式仕様記述言語 VDM-SL を実行するための VM である．ViennaVM の開発では，VM の仕様は VM の実装の処理速度および信頼性の両方に大きな影響を与えると考え，仕様の品質を向上するためのアプローチとして VDM-SL による仕様記述を行なった．本論文では，2 節で VDM-SL の簡単な概要を説明し，3 節で ViennaVM のレジスタに関する特徴を説明し，4 節でレジスタへのアクセスおよび演算命令の VDM-SL 仕様と C 言語での実装を示す．5 節で，仕様記述工程および実装工程への影響についてテストを中心に議論し，また，現在の実装での性能を評価する．最後に，6 節で全体をまとめるとともに，今後の課題を示す．

2. VDM-SL の概要

本節では，ViennaVM の形式仕様記述言語として採用した VDM-SL について概要を説明する．VDM-SL はモデル規範型の形式仕様記述言語であり，仕様記述者は対象をモジュールに分解して，各モジュールが持つ機能を内部状態および内部状態に対する操作としてモデル化する [5].

VDM-SL では，システムをモジュール化し，各モジュールをデータ型，定数，関数，状態空間，操作の 5 種類の定義によって記述する．各モジュールの外部インターフェイスは `exports` 文によって宣言され，外部モジュールが定義する要素への参照は `imports` 文によって宣言する．各モジュールではデータ型を `types` 部で宣言し，定数および関数をそれぞれ `values` 部および `functions` 部で定義することによって，そのモジュールが扱う概念とその関係を記述する．VDM-SL では定数および関数は参照透明である．`state` 部では，モジュールが持つ内部状態を定義するために，静的に型付けられた変数のリスト，状態が常に守るべき不変条件および初期状態を宣言する．モジュールが提供する操作を `operations` 部で定義する．操作はプログラミング言語の手続きに相当し，静的に型付けされている．

VDM-SL は実行可能なサブセットを持つ．VDM-SL で記述された仕様は，関数や操作について評価する式や文を陽に記述し，かつ，`forall` などの限量子や有限集合の内包表記など，全要素の評価が必要な式について，型ではなく有限集合に束縛することで，インタプリタ等で実行することができる．VDMTools[6]，The Overture tool[7, 8]，ViennaTalk[9] などの VDM-SL 開発環境はインタプリタを持ち，ユーザは記述中の仕様をインタプリタ実行することで記述対象が持つ振る舞いや特性を理解することができる．さらに，vdmUnit や ViennaUnit[10] などの単体テストフレームワークを利用して，VDM-SL 仕様を継続的に単体テストすることが可能である．実際の開発適用事例においても，VDM-SL のオブジェクト指向拡張である VDM++[11] による仕様記述に対するテストが行われ，仕様記述の品質向上および成果物の信頼性向上に高い成果を上げている [12].

3. ViennaVM の特徴

ViennaVM は VDM-SL 仕様を実行することを目的とした VM で，VDM-SL 仕様から自動生成された IR コードを，IoT を含む多様なプラットフォーム上で実行することを想定している．VDM-SL 仕様からの実装の自動化として，VDM-SL から C++，Java，Smalltalk などの多様なホスト言語を対象とするプログラムソース自動生成器が研究され開発されている．プログラムソースの自動生成器は実装工程のコストを劇的に削減する一方，ホスト言語の言語仕様および処理系の更新に追従して保守する必要がある．VDM-SL の実行に必要な機能を持つ ViennaVM の仕様を厳密に記述し，様々なプラットフォーム上でそのプラットフォームに適したプログラミング言語で ViennaVM を実装することで，VDM-SL から単一の IR コード自動生成器によって多様なプラットフォーム上で動作する実装を得ることが ViennaVM 開発の目標である．

図 1 に ViennaVM のモジュール構成を示す．ViennaVM は，ViennaVM が扱うデータフォーマットを定義する Data モジュール，レジスタおよびレジスタに対する読み書きを行う Register モジュール，ヒープメモリの管理および

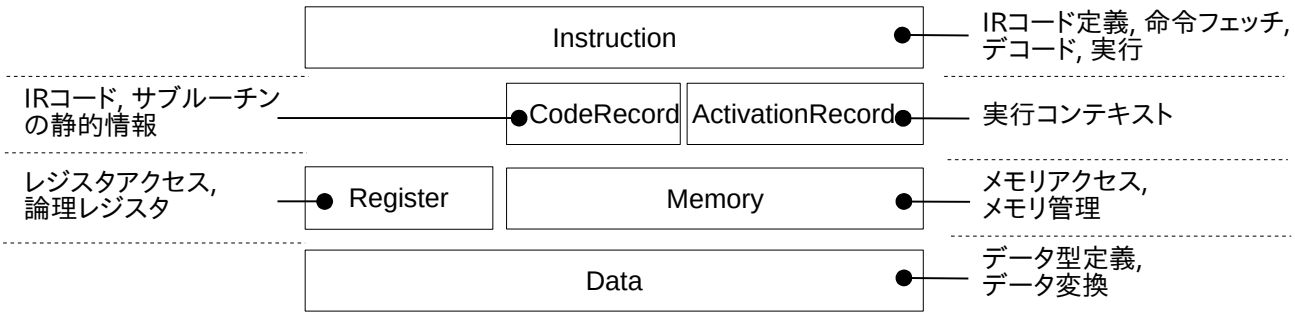

図 1. ViennaVM のモジュール構成

ヒープメモリに対する読み書きを行う Memory モジュール，コードブロックを管理する CodeRecord モジュール，実行コンテキストを管理する ActivationRecord モジュール，および，コードブロックから IR コードを読み出し実行する Instruction モジュールから成る．本論文では，ViennaVM のモジュールの中でも特に ViennaVM に特徴的な設計となっている Register モジュールに着目し，その特徴を説明する．

3.1. 安全で効率的なタグ付けとタグ除去

VDM-SL は静的型付け言語であるが，実行には実行時型情報が必要である．ViennaVM では，データオブジェクトを 64 ビットのタグ付きワードによって表現する．具体的には，63 ビット符号付き整数，32 ビット浮動小数点数，32 ビットユニコード文字，および 56 ビットポインタを 64 ビットのタグ付きポインタに符号化する．

データオブジェクトとしてタグ付きワードを採用する VM の多くは，メモリやレジスタにタグ付きワードを格納し，算術演算などタグを除去しての演算が必要な場合にのみ，演算処理を行う時に一時的にタグを除去したデータを取り出し，演算を行い，結果をタグ付きワードに変換する．この方式では，メモリ上のデータ表現をタグ付きワードに統一することで，タグ付きワードとタグを除去されたデータを混同する危険を減少させ，安全にデータを扱うことができる．一方で，複雑な算術演算を行う時には多くのタグ付けとタグ除去が実行されるという欠点がある．

冗長なタグ付けとタグ除去を回避するために，明示的にタグ付けおよびタグ除去を行う IR コードを定義するアプローチもある．IR コードが明示的にタグ付けおよびタグ除去を行う場合には，タグ除去されたデータオブジェクトの正当性は IR コードに依拠することになる．例えば，整数値を持つタグ付きワードをタグ除去し，演算を行った結果をポインタとしてタグ付けすることが可能になると，不正なアドレスへのメモリ書き込みが可能になる．

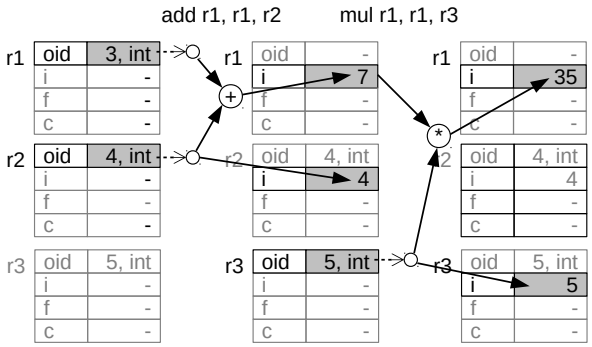

図 2. レコード型のレジスタによる自動タグ除去と演算の例

ViennaVM では，冗長なタグ付けとタグ除去を削減するために，タグ付与とタグ除去をレジスタ内部で処理するよう設計した．レジスタをタグ付きワード，符号付き整数，浮動小数点数，ユニコード文字の 4 つのフィールドを持つレコードとして，算術演算命令やメモリアクセスなどで利用される時にのみタグの除去およびタグの付与が実行され，その結果がフィールドに格納される．その後，同じレジスタに対する算術命令が実行される時には，フィールドに格納されたタグなしデータオブジェクトを直接読み込んで算術演算が実行される．これによっ

て，冗長なタグ付けの回避と安全なタグ付けおよびタグ除去を両立させている．

図 2 に ViennaVM の演算命令における自動的なタグ除去の例を示す．2 つの命令 `add r1, r1, r2` および `mul r1, r1, r3` によって，`r1 ← (r1 + r2) * r3` を実行するためのタグ除去と算術計算の手順を説明する．従来の，算術命令で常にタグ除去を行い，算術演算の結果にタグ付与を行う場合，各算術命令の前に 2 つのオペランドに対するタグ除去と，各算術命令後に 1 回のタグ付与が行われ，合計すると 4 回のタグ除去と 2 回のタグ付与が実行される．ViennaVM での自動的なタグ除去およびタグ付与では，3 回のタグ除去で同様な算術結果が得られる．

まず左端に初期状態として，3 つのレジスタ `r1`, `r2`, `r3` に，それぞれ整数 3, 4, 5 がメモリからロードされた状態を仮定する．ここでまず，命令 `add r1, r1, r2` を実行する．この命令は，2 つのレジスタ `r1` と `r2` を加算した結果を `r1` に格納する．レジスタ `r1` にはタグ付きワードフィールド（oid）のみ格納されているため，まずはタグを除去して整数 3 を得る．レジスタ `r2` も同様で，タグを除去して整数 4 を得る．加算した結果である整数 7 をレジスタ `r1` の整数フィールド（i）に格納する．また，レジスタ `r2` の整数フィールド（i）にもタグを除去した結果である整数 4 を格納する．

次に，命令 `mul r1, r1, r3` を実行する．この命令は，2 つのレジスタ `r1` と `r3` を乗算した結果を `r1` に格納する．レジスタ `r1` には整数フィールド（i）に 7 が格納されているため，これをそのまま利用する．レジスタ `r3` にはタグ付きワードフィールド（oid）のみ格納されているため，まずはタグを除去して整数 5 を得る．乗算した結果である整数 35 をレジスタ `r1` の整数フィールド（i）に格納し，レジスタ `r3` の整数フィールド（i）にもタグを除去した結果である整数 5 を格納する．

以上の手順で 2 つの算術演算命令が実行されるが，この中でタグ除去は 3 回のみ行われる．さらに，この後でレジスタ `r2` やレジスタ `r3` の整数値が必要な算術演算命令を実行する場合には，タグ除去なしで整数値を取り出すことができる．レジスタ `r1` にはタグ付きワードフィールド（oid）に値が格納されていないが，タグ付きワードが必要な命令を実行する時点でタグ付与を行なってタグ付きワードフィールドを得る．ViennaVM は以上の方法で，タグ付けされたデータに対して安全かつ少ないオーバーヘッドによる演算処理を実現している．

3.2. レジスタ空間

JVM を初めとする多くの VM はスタックマシンであり，手続きへの引数および返値の受け渡しはスタックを介して行われる．レジスタの内容を引数または返値としてメモリ上のスタックに書き込むと，第 3.1 節で説明した自動的にタグ除去されたデータオブジェクトが失われてしまう．また，ViennaVM では参照カウントによる自動ガーベージコレクション (自動 GC) を採用している．コピー GC など，参照を追跡していくことで参照のないメモリブロックを発見する他の自動 GC アルゴリズムに比べて，参照カウントによる自動 GC はカウンタの増減という比較的小さなオーバーヘッドで参照のないメモリブロックを自動的に発見し解放することができる．しかし，手続きへの引数や返り値としてポインタ値を保持しているレジスタをメモリ上のスタックに書き込むと，参照カウンタの増減処理が必要になり，メモリ管理のオーバーヘッドを増加させる．ViennaVM では，メモリ上のスタックを介したデータの受け渡しを避けるために，手続きへの引数および返値の受け渡しはレジスタを介して行う．

引数および返値の受け渡し以外に，スタックはレジスタの退避場所としても利用されることがある．一般に多くの VM では，スタックを使ってレジスタの内容の退避を行う．ViennaVM ではレジスタは 4 つのフィールドからなるレコードとして定義されるため，レコード全体をスタックに退避させレジスタに復元すると，データトラフィックが大きくなり，性能面で不利である．一方，ViennaVM ではヒープメモリ上のメモリブロックにはタグ付きワードのみを格納する．スタックにレジスタ内のデータのうちタグ付きワードを退避復元させる場合には，タグ付けおよびタグ除去の回数が増加し，これも性能面で不利になる．

ViennaVM では，IR コードのオペランドとして指定されるレジスタ番号にオフセットを加えたものを物理的なレジスタ番号として，オフセットを変更することでレジスタの退避および復元を行う．図 3 にレジスタ番号のオフセットによるレジスタ退避の例を示す．ViennaVM のサブルーチンは，そのコードが利用するレジスタ数を保持する．初期状態として，4 つのレジスタを使うサブルーチンをレジスタオフセット値が 3 で実行しているとする．この時，サブルーチン内では r1, r2, r3, r4 の 3 つの論理レジスタ名を利用することができ，それらはそれ

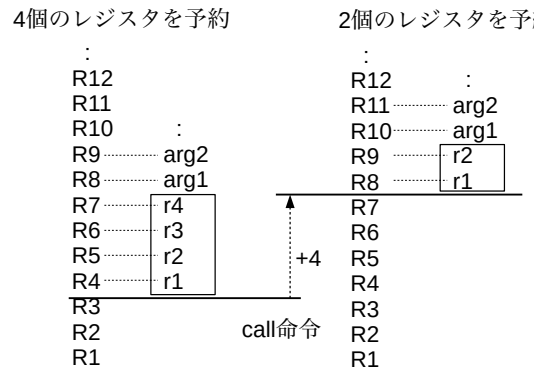

図 3. 論理レジスタ番号によるレジスタ退避の例

ぞれ実レジスタ R4, R5, R6, R7 を指す．このサブルーチンから，レジスタを 2 つ使う別のサブルーチンを `call` 命令で呼び出す時，引数は r5 相当の実レジスタ R8, r6 相当の実レジスタ R9 で渡される．`call` 命令はレジスタオフセットを呼び出し側のレジスタ数分増加させる．この例では，+4 増加させる．するとレジスタオフセットは 7 となり，呼び出された側のサブルーチンでは論理レジスタ名 `r1` および `r2` はそれぞれ実レジスタ `R8` および `R9` を指す．すなわち，第 1 引数として渡されたデータをレジスタ名 `r1` として受け取る．

サブルーチンからの復帰を行う `RET` 命令は，レジスタオフセットを復帰先のレジスタ数分だけ減少させるとともに，復帰元で使用していたレジスタの内容を初期化することで参照カウンタの増減処理を行う．

このレジスタ退避の方式の利点は，メモリトラフィックの節約である．例えば手続き呼び出しによってレジスタを 10 個退避し，後で復元する場合，レジスタレコード単位で退避する方式では 640 バイトのメモリ間データ転送が必要となる．タグ付きワードのみを退避する場合には，160 バイトのメモリ間データ転送と，最大 10 回のタグ付けと最大 10 回のタグ除去が必要となる．ViennaVM が採用するレジスタオフセットの場合，オフセット時のホスト言語での整数型の加減算のみでレジスタの退避および復元が可能である．

ViennaVM は実レジスタ番号および論理レジスタ番号ともに 16 ビットのレジスタ空間を持つ．Dalvik VM[3] をはじめとする広いレジスタ空間を持つレジスタマシン方式の VM では，豊富なレジスタ数を生かしてサブルーチン呼び出しのインライン展開をより深く行うような最適化を可能にしている．ViennaVM では，レジスタ番号をオフセットすることで，インライン展開をしなくても豊富なレジスタ数を有効に利用することができる．

4. ViennaVM の仕様と実装

本節では，ViennaVM の仕様と実装について，3.1 節および 3.2 節で説明したレジスタへのデータの読み書きおよび論理レジスタ番号に関する VDM-SL 仕様と C 言語による実装を説明する．

4.1. レジスタに対するデータの読み書き

3.1 節で説明した通り，ViennaVM では安全かつ効率的なタグ付きワードの扱いのために各レジスタに静的型ごとのフィールドを持たせ，必要に応じてタグ付けおよびタグ除去を行い，その結果を次の使用時に備えて適切なフィールドに格納する．これは ViennaVM 独自の設計であり，その正確な機能仕様を自然言語および表などの前形式的な表現で定義するだけでは解釈が一意に定まらない危険がある．また，解釈が一意に定まる場合でも，仕様に誤りがある場合に，自然言語などによる記述では誤りの発見が困難であり，しばしば実装後に誤りが発見され，手戻りの原因になり得る．

また，ViennaVM でのレジスタの読み書きには，タグ付けおよびタグ除去だけでなく，型変換も伴う場合がある．VDM-SL では整数型や自然数型が実数型の部分型として定義されている．VDM-SL の言語仕様では実数型の内部表現および精度に関する規定はなく，全ての整数値は実数型の値であり，`1.0` と `1` は同じ値を意味する．例えば，式 `1 = 1.0` は `true` である．

そこで，ViennaVM では，整数値が格納されたレジスタから浮動小数点数を読み出す時に，読み出し処理の中で整数値から浮動小数点数に変換した値が得られるものとした．本節ではそれが VDM-SL によってどのように定義されているのかを説明する．

図 4 に ViennaVM 内部でのデータ型定義の一部を抜粋する．`OID` 型がタグ付き 64 ビットワード，`Int` 型は 63 ビット符号付き整数，`Float` 型は 32 ビット浮動小数点数，`Char` 型は 32 ビットユニコード，`Pointer` 型は

```
types
  Qword = nat
    inv qw == qw < 0x10000000000000000;
  OID = Qword;
  Int = int
    inv i == (i > -0x8000000000000000
      and i < 0x8000000000000000)
    or i = invalidIntValue;
  Float ::
    sign : nat
    exponent : nat
    fraction : nat
        inv mk_Float(s, e, f) ==
          (s < 2 and e < 0x100) and
          f < 0x1000000;
  Char = nat
    inv f == f < 0x100000000
      or f = invalidCharValue;
  Pointer = nat
    inv p == p < 0x100000000000000
      or p = invalidPointerValue;
```

図 4. VDM-SL 仕様でのデータ型定義の抜粋

```
types
  Reg ::
    oid : Data`OID
    i : Data`Int
    f : Data`Float
    c : Data`Char;
  Register = nat;
```

図 5. レジスタの定義

56 ビットポインタとして定義されている．具体的な定義としては，タグ付きワード型 OID や整数型 Int がそれぞれ自然数型および整数型に不変条件を与えることによって定義している．また，浮動小数点数型 Float をレコード型で定義している．

図 5 に Register モジュールでのレジスタの定義を示す．レコード型 Reg が，oid，i，f，c の 4 つのフィールドから構成されるレコード型として定義されている．それぞれのフィールドには，図 4 で定義された OID 型，Int 型，Float 型，Char 型が静的につけられている．また，レジスタ番号を表す型 Register が自然数として Register モジュールで定義されている．

レジスタにデータを書き込む時には，まずは論理レジスタ番号から実レジスタ番号を求め，該当するレジスタの書き込みデータの型に対応するフィールドに書き込み，それ以外のフィールドを無効値に設定する．例として，レジスタに整数値を書き込む操作 write_int の仕様を図 6 に示す．5 行目はレジスタオフセットで論理レジスタ番号をオフセットして実レジスタ番号 r を求めている．8-9 行目は参照カウントによるメモリ管理に関する記述であり，書き込み前のレジスタがポインタであった場合に，書き込みによりその参照が失われるためメモリブロックの参照カウントを減らす．10 行目以降で，i フィールドに整数値を，それ以外のフィールドには無効値を書き込む．

```
operations
  write_int : Register * Data`Int ==> ()
  write_int(dst, i) ==
    let
      r : Register = baseRegister + dst,
      p = registers(r).oid
    in
      (if Data`isPointer(p)
      then Memory`decrement_reference_count(p);
      registers(r) := mk_Reg(
        Data`invalidTag,
        i,
        Data`invalidFloatValue,
        Data`invalidCharValue));
```

図 6. レジスタへの整数値の書き込みの仕様

レジスタからデータを読み出す時には，まず対応するフィールドの値が無効値でなければその値を返す．無効値であれば，互換性のある型のフィールドから型変換をしてフィールドに書き込んだ上で，読み出し値とする．

例として，レジスタから整数値を読み出す操作 read_int の仕様を，関連する 2 つの操作とともに図 7 に示す．操作 basic_read_int は，指定されたレジスタの i フィールドに値が格納されていたらそれを返し，さもなくば oid フィールド（タグ付きワード）に格納された値を整数型に型変換した結果を i フィールドに書き込んだ上で，それを返す．操作 basic_read_float は同様のことを f フィールドに対して行う．

前述のように，VDM-SL では，整数型 int および自然数型 nat および nat1 は実数型 real の部分型であり，例えば 1.0 は実数であるだけでなく整数型の値でもある．つまり，式 1 = 1.0 は true である．そこで ViennaVM では，浮動小数点数が格納されたレジスタからも，それが整数でもある場合には整数値として読み出し可能とした．整数 basic_read_int 操作は，指定されたレジスタから basic_read_int 操作で整数値を読み出し，もし結

```
operations
  basic_read_int : Register ==> Data`Int
  basic_read_int(src) ==
    let
      r : Register = baseRegister + src,
      reg : Reg = registers(r),
      i : [Data`Int] = reg.i
    in
      (if i <> Data`invalidIntValue
      then return i;
      let i2 : Data`Int = Data`oid2int(reg.oid)
      in
        if
          i2 <> Data`invalidIntValue
        then
          (registers(r) .i := i2;
          return i2);
      return Data`invalidIntValue);

  basic_read_float : Register ==> Data`Float
  basic_read_float(src) ==
    let
      r : Register = baseRegister + src,
      reg : Reg = registers(r),
      f : [Data`Float] = reg.f
    in
      (if f <> Data`invalidFloatValue
      then return f;
      let
        f2:Data`Float=Data`oid2float(reg.oid)
      in
        if
          f2 <> Data`invalidFloatValue
        then
          (registers(r) .f := f2;
          return f2);
      return Data`invalidFloatValue);

  read_int : Register ==> Data`Int
  read_int(src) ==
    (let i : Data`Int = basic_read_int(src)
    in
      if i <> Data`invalidIntValue
      then return i;
    let f : Data`Float = basic_read_float(src)
    in
      (if
        f <> Data`invalidFloatValue
      then
        (let
          r = Data`float2real(f),
          int_r = floor (r + 0.5)
        in
          (if r = int_r
          then
            (registers(baseRegister+src).i :=
              int_r;
            return int_r))));
    return Data`invalidIntValue);
```

図 7. レジスタからの整数値の読み出しの仕様

```
  idiv : Register`Register * Register`Register
         * Register`Register ==> ()
  idiv(dst, src1, src2) ==
    if
      (dst > 0 and src1 > 0) and src2 > 0
    then
      let
        int1:Data`Int=Register`read_int(src1),
        int2:Data`Int=Register`read_int(src2)
      in
        if
          int1 <> Data`invalidIntValue
            and int2 <> Data`invalidIntValue
        then
          Register`write_int(dst, int1 div int2)
        else
          err("idiv: operands not integer")
    else
      err("idiv: operands not specified");
```

図 8. idiv 命令の仕様

果が未定義値であった場合には，basic_read_float 操作で浮動小数点数値を読み出し，もし結果が未定義値でなく，かつ，整数値であったら，Int 型に型変換した結果を i フィールドに書き込んだ上で，それを返す．

上記の浮動小数点数から整数への型変換は，ViennaVM が VDM-SL に特化した VM であることから有用であって，浮動小数点数 1.0 と整数 1 が異なる数値として扱われる他の多くのプログラミング言語では不要である．VDM-SL で仕様を記述することで，上記のような複雑な内容を曖昧さなく厳密に記述することができる．

write_int および read_int 操作を使った IR コードの仕様の例として，図 8 に idiv 命令の仕様を示す．第 2 オペランドおよび第 3 オペランドとして指定されたレジスタから Register モジュールの read_int 操作で整数値をそれぞれ読み出し，両方のレジスタから正常に整数値が読み出せた場合に整数除算の結果を第 1 オペランドとして指定されたレジスタに書き込む．

idiv 命令の C 言語での実装を図 9 に示す．図 8 の VDM-SL 仕様に忠実な実装であることがわかる．

5. 議論

本節では，VM の開発に VDM-SL を導入したことの影響を議論する．VM に要求される品質項目で特に重視されるものとして，信頼性と性能に着目し，5.1 節でテストやデバッグについて，5.2 で性能面について論じる．

```
void idiv(Register dst, Register src1,
          Register src2) {
  if (dst && src1 && src2) {
    Int int1 = read_int(src1);
    Int int2 = read_int(src2);
    if (int1 != invalidIntValue
        && int2 != invalidIntValue) {
      write_int(dst, int1/int2);
      return;
    }
    err("idiv: operands not integer");
  } else {
    err("idiv: operands not specified");
  }
}
```

図 9. idiv 命令の C 言語での実装

5.1. テストおよびデバッグ

ViennaVM の VDM-SL 仕様で定義された関数および操作は，ユニットテストフレームワーク viennaUnit [4] を使ってテストされた．図 10 に，`Register` モジュールに対応するテストモジュール `RegisterTest` の抜粋を示す．ViennaVM の各モジュールに対応してテストモジュールを記述し，仕様を修正する度に自動的に全テストモジュールを実行した．また，VDM-SL で記述されたテストモジュールは C 言語に移植され，それを C 言語での実装に対する単体テストとして利用した．表 1 に VDM-SL 仕様および C 言語での実装およびテストモジュールの行数およびテストケース数をまとめた．

ViennaVM の開発では，VDM-SL 向けユニットテストフレームワーク ViennaUnit を利用した．ViennaUnit は，仕様を編集し保存する度に，IDE がバックグラウンドタスクとして全てのテストケースを実行し，テストが失敗した場合にのみデスクトップ上に通知を表示する機能を提供している．VDM-SL 仕様を記述したファイルへの編集が保存されると，ViennaUnit はモジュール名が `Test` で終わるモジュールをテストモジュールと見なし，テストモジュール内に定義された操作のうち操作名が `test` で始まる操作をテストケースと見做して実行する．この機能を利用することで，仕様記述者が明示的にユニットテストを実行するよりも，仕様記述者の負担を増やすことなく高頻度でテストを実行し，仕様の誤りを早期のうちに発見することができた．また，デバッグのための仕様修正に対しても高い頻度でテストが自動的に実行されたため，修正によるリグレッションを早期発見し，再修正することができた．

```
values
  int1 : Data`Int = 0x1234;

operations
  testWriteBoxedIntAndUbox : () ==> ()
  testWriteBoxedIntAndUbox() ==
    (Register`write_oid(1, Data`int2oid(int1));
    assertEquals(
      Register`read_int(1),
      int1,
      "int->int");
    assertEquals(
      Register`read_float(1),
      Data`real2float(int1),
      "int->float");
    assertEquals(
      Register`read_char(1),
      Data`invalidCharValue,
      "int-char");
    assertEquals(
      Register`read_pointer(1),
      Data`invalidPointerValue,
      "int->pointer"))
```

図 10. RegisterTest モジュールの抜粋

ViennaUnit は，表明検査のための操作を 2 つ提供している．`assert` 操作は，引数を 2 つ取り，第 1 引数が true であるかどうかを確認し，true でない場合に第 2 引数をエラーメッセージとしてテスト失敗とする．`assertEquals` 操作は，引数を 3 つ取り，第 1 引数と第 2 引数が等しいかどうかを確認し，等しくない場合に第 3 引数をエラーメッセージとしてテスト失敗とする．テストケースでは上記 2 つの表明検査を使って，テスト対象の動作を確認する．

ViennaVM での開発では，仕様の誤りの多くはデータ型に付与された不変条件に対する違反として検出された．テストケース内での `assert` 操作および `assertEquals` 操作はテスト対象の動作結果に対してのみ検査するのに対して，データ型に付与された不変条件はテスト対象となる動作の中間状態も含め継続的に検査が行われる．多くのテストケースでは，テストケースに直接記述された表明検査よりも，多くの種類のデータ型不変条件が，多くの回数で，テストケースの表明検査よりも先に検査されることが，データ型不変条件が仕様の誤りの発見に有効であった理由と考えられる．

C 言語での実装のユニットテストは，VDM-SL によるテストモジュールを C 言語に移植することで実装した．図 11 にレジスタへのデータアクセスのテストの一部を

表 1. 仕様および実装の行数およびテストケース数

モジュール名	VDM-SL 仕様の行数	VDM-SL テストの行数	C 実装の行数	C テストの行数
Data	279	209	138	57
Register	255	147	254	204
Memory	328	253	291	233
CodeRecord	100	105	127	119
ActivationRecord	63	29	50	31
Instruction	657	473	649	455
合計	1760	1131	1589	1136

```
static Int int1 = 0x1234;

static int testWriteBoxedIntAndUbox() {
  write_oid(1, int2oid(int1));
  assertEquals(
    read_int(1),
    int1,
    "int->int");
  assertEquals(
    read_float(1),
    (Float)(real2float((float)int1)),
    "int->float");
  assertEquals(
    read_char(1),
    invalidCharValue,
    "int-char");
  assertEquals(
    read_pointer(1),
    invalidPointerValue,
    "int->pointer");
  return 0;
}
```

図 11. Register モジュールの C 言語でのテストコードの抜粋

抜粋する．C 言語での実装に対するテストで発見された誤りの多くは実装上の誤りであった．例えばレジスタへのアクセスにおいてポインタを使って計算量を節約して発生した誤りが発見された．一方で，一般のプログラミングで発生する境界条件の誤りや場合分けの抜けなどは，現在のところ見つかっていない．仕様に対するテストで計算式やアルゴリズムの誤りが発見され修正されていたためであると考えられる．

形式手法を導入した開発では，仕様記述の工数が増加する一方で実装の工数が大幅に短縮されるフロントローディング効果が報告されている．VM 開発でもテスト工程で要する工数が仕様記述工程に前倒しされたと考えられる．VDM-SL の実行可能な仕様は，プロトタイプやリファレンス実装でのプログラムに近いように見えるが，システムが取りうる状態に対する制約条件を記述するモデル規範型の形式仕様記述言語としての言語機能が，機能定義の誤り発見と実装上の詳細の誤り発見を分離し，フロントローディング効果につながったと考えられる．

5.2. 性能

一般に，言語処理系のランタイムとしての VM に対する要求として，性能への優先順位は高い．VM を開発するにあたって，要求される性能が実現可能であるかどうかは，常に重大な関心事である．本節では，ViennaVM の開発に VDM-SL を導入した結果として，ViennaVM の C 言語での実装の性能をフィボナッチ数の計算で計測し，それを VDM インタプリタ VDMJ と Python3 と比較した．

表 2 にその計測結果をまとめた．実行環境として，サーバ機やデスクトップ PC 向けに広く普及している x64 アーキテクチャの Core i5 プロセッサと，組み込みなどで広

表 2. fib(30) による性能ベンチマーク

プロセッサ	実行時間 (ミリ秒)	
	ViennaVM	Python3
Core i5	348.7	466.7
ARMv7	5,107.1	6,140.6

Core i5: 2.5GHz (macOS Mojave, 64 ビット)
ARMv7: 1.4GHz (Raspbian Stretch, 32 ビット)

く普及している ARMv7 プロセッサを対象とした．各プロセッサ上で，VDMJ，ViennaVM，ベンチマークとして 30 番目のフィボナッチ数を計算するのに要する時間をそれぞれの処理系で 10 回計測し，その平均を求めた．いずれのプロセッサでも，Python 3 を上回る性能を示した．ただし，既に長く実用されている Python 3 に対して，ViennaVM は開発途上の段階であり，これからコード量が増加すると性能の劣化が起こる可能性がある．

6. まとめ

ViennaVM の開発に VDM-SL を導入し，C 言語による実装を行なった．VM の仕様の多くの部分は，VM の動作を記述するための，VM が扱うデータ型の定義や各 IR コードの処理内容の操作的な意味を与えることに占められる．VDM-SL を導入することで，VM の仕様を曖昧さなく記述し，仕様の誤りの発見と実装の誤りの発見を明確に分離することができた．また，性能面でも広く利用されているプログラミング言語インタプリタと同等の速度を実現できた．これにより，VM 開発に対しても VDM-SL の有効性を示すことができた．

ViennaVM の開発は継続中であり，VDM-SL の実行可能なサブセット全体に必要な機能を完成させた上で，JIT コンパイルや IR コード書き換えによる最適化などを仕様記述し実装する予定である．

参考文献

[1] A. Rigo and S. Pedroni, “PyPy’s Approach to Virtual Machine Construction”, In companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA ’06), pp. 944-953, 2006.

[2] T. Lindholm, et al, “ The Java Virtual Machine Specification”, Addison-Wesley Professional, 2014.

[3] D. Ehringer, “The dalvik virtual machine architecture”, available at `http://show.docjava.com/posterous/file/2012/12/10222640-The_Dalvik_Virtual_Machine.pdf`, 2010.

[4] T. Oda, K. Araki and P. G. Larsen, “ViennaVM: a Virtual Machine for VDM-SL development”, In proc. of the 16th Overture Workshop, pp. 39-56, 2018.

[5] J. Fitzgerald and P. G. Larsen “Modelling Systems – Practical Tools and Techniques in Software Development”, Cambridge University Press, 1998.

[6] J. Fitzgerald, P. G. Larsen and S. Sahara “VDMTools: advances in support for formal modeling in VDM”, ACM Sigplan Notices, Vol. 43, No. 2, pp. 3–11, 2008.

[7] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl and M. Verhoef “The Overture Initiative – Integrating Tools for VDM”, *SIGSOFT Softw. Eng. Notes*, Vol. 32, No. 1, pp. 1–6, 2010.

[8] P. G. Larsen, K. Lausdahl, P. W. V. Tran-Jørgensen, A. Ribeiro, S. Wolff and N. Battle “Overture VDM-10 Tool Support: User Guide”, The Overture Initiative, TR-2010-02, 2010.

[9] 小田朋宏, 荒木啓二郎 “形式仕様工程の初期段階に着目した統合仕様記述環境 ViennaTalk”, コンピュータソフトウェア, 34 巻, 4 号, ppp. 4_129-4_143, 日本ソフトウェア科学会, 2017.

[10] 小田朋宏, 荒木啓二郎 “アジリティのある探索的形式仕様記述のためのテストフレームワーク”, ソフトウェアシンポジウム 2018 論文集, 2018.

[11] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat and M. Verhoef “Validated Designs For Object-oriented Systems”, Springer, 2005.

[12] T. Kurita and Y. Nakatsugawa “The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip”, *Intl. Journal of Software and Informatics*, Vol. 3, No. 2-3, pp. 343–355, 2009.