修 士 論 文

論理学的アプローチによるJAVA仮想機械の諸性質の分析および実装への応用

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

樋口 智之

2002年3月

修 士 論 文

論理学的アプローチによるJAVA仮想機械の諸性質の分析および実装への応用

指導教官　大堀淳 教授

審査委員主査　大堀淳教授
審査委員　田島敬史 助教授
審査委員　小野寛晰 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

010093 樋口 智之

提出年月: 2002 年 2 月

概 要

本論文では JAVA バイトコードに対する型システムを提案し、型システムの健全性を証明する。さらに型推論アルゴリズムを構築してその実装を行なう。本論文が示す型システムはこれまで提案されてきたものとは異なり、プログラム自体を型付けする。すなわち、プログラムの実行時の振舞を静的に検証することが可能である。ラムダ計算の型システムと同様の概念に基づき定義されているため、そこで発展してきた種々の型理論を適用することが可能である。例えば、プログラムをラムダ計算の多相的 let 式に似た項とみなすことでバイトコードベリファイアを型推論アルゴリズムとして記述できる.

目 次

図 目 次

表 目 次

第1章　序論

1.1　JAVAの誕生

インターネットの出現により人々の生活は一変した. 人々は家庭にいながらにして世界中のコンピュータにアクセスして様々な情報を得ることができるようになったのである. インターネットのようなアーキテクチャの異なる機械が相互に接続された大規模なネットワークに要求されることは, いかにこれらの互換性のないコンピュータを統一的に扱うことができるかということである. つまりこれはこれらのコンピュータをネットワークを越えて制御できるソフトウェアを構築することは可能かということを意味する.

ソフトウェアを構築するためにはそれを記述するためのプログラミング言語が必要となる. しかし従来からよく使われてきた言語, 例えばC言語などは基本的に個々のマシンに依存するうえ, ネットワーク機能が貧弱であるため, ネットワーク上でのソフトウェアの開発が困難であった. ネットワークが急速に発達するにつれてこの問題はますます大きなものとなり従来の言語を拡張してネットワーク機能を付加する試み行なわれたが, 信頼性が低く, 依然マシン依存という問題点も解消されなかった. そこで当然のごとく全く新しいプログラミング言語を作ろうという動きが出てきた. そのような新しい言語に求められることはネットワークに対応にしていることおよびマシン非依存であるということである. この二つの要求を満たすプログラミング言語として生まれたのが, 今最も注目を浴びている言語 JAVA である.

1.2　JAVAとJVM

JAVA はその設計段階から異機種間の分散ネットワーク上での使用を意図されたプログラミング言語である. その最大の特徴は, 記述したプログラムはどのようなマシンでも同じように動作することであり, これを支えるのは JAVA 仮想マシン (JVM) と呼ばれるソフトウェアから構成された仮想的なハードウェアである. 実際のマシン同様に JVM は JAVA バイトコードと呼ばれるマシン語を解釈実行することが可能である. JAVA コンパイラは JAVA プログラムをバイトコードから構成されるクラスファイルに変換する. クラスファイルは JVM における実行ファイルに相当するため, 一度 JAVA プログラムをコンパイルすれば JVM を搭載したマシンならばプラットホームに関わらず JAVA プログラムを実行することが可能である. この構造により JAVA プログラムの高い可搬性が実現された.

このような特徴から JAVA は現在のインターネット社会で爆発的にユーザを獲得した. 最もよく知られるネットワーク上の JAVA アプリケーションは Web アプレットと呼ばれる Web ブラウザ上で動作する小さなプログラムである. インターネットのユーザはリモートサイトの WWW サーバから自分のマシンにクラスファイルをダウンロードして Web ブラウザに組み込まれた JVM で直接実行することが可能である.

1.2.1　セキュリティ上の問題

しかし, この”リモートサイトからダウンロードしたクラスファイルを直接実行出来る”ということが大きなセキュリティ上のリスクを生み出すことになった. 例えば, ユーザがリモートサイトからダウンロード

したクラスファイルが Sun 純正の JAVA コンパイラによって生成されたクラスファイルであるという保証はない. クラスファイルは専用のツールを使えば, 少し知識のあるものにとっては簡単に作成することが出来るため, クラッカーによって生成された悪意のあるクラスファイルである可能性がある. そのようなクラスファイルを実行してしまうと貴重なデータが改算あるいは漏洩されたり, 最悪の場合マシンが破壊されるという事態に陥るかもしれない. また例え正しいクラスファイルであっても, ダウンロード中にクラスファイルの一部が欠落しているということも考えられる.

1.2.2 バイトコードベリファイア

このような危険性を排除するため JVM ではクラスファイルを実行する前に, それが JVM が要求する仕様を満たしているかを静的に検証する. もし検証に失敗すると, そのクラスファイルは正当でないと判断され実行されることはない. このクラスファイルの検証を行なう部分はベリファイアと呼ばれ, 大きく分けて次の４つのパスに分けることができる.

1. ロード時, クラスファイルの形式が仕様を満たすかどうかを検証する
2. リンク時, バイトコード配列を除いたクラスファイルの内容を検証する
3. バイトコード配列の検証する
4. 他のクラスファイルを参照する際に必要となる検証

この中で 3 のバイトコード配列の検証が最も複雑かつ重要なプロセスであり, バイトコードベリファイアと呼ばれる. バイトコード配列とは JAVA プログラムのメソッド本体に対応する部分であり, JAVA でメソッドを呼び出すことはクラスファイル中の対応するバイトコード配列を JVM で実行することに他ならない. バイトコードベリファイアはコード配列のデータフロー分析を行ない, 次のような種々の制約を満たしているかをチェックする.

- ジャンプ命令の分岐先が正しいか？
- 初期化されていないローカル変数へのアクセスがないか？
- ローカル変数およびスタックの内容は適当か？

このような制約条件を検証することにより, バイトコードの安全性の向上が図られている.

1.3 JVM の問題点と既存の研究

上で述べたように, バイトコードベリファイアは JVM を構成する最も重要な部分の一つであり, 細部まで慎重に設計されている. ところがそのアルゴリズムはかなり複雑であるにも関わらず, その仕様 [10] となる記述は自然言語で書かれているためあいまいな部分が多く, またそれが本当に理論的に正しいのかという数学的な証明はされていない. この問題点は早くから指摘され多くの研究者が JVM を形式的に記述しようと試みてきた. 多くが JVM に対する型システムの導入に関する研究である.

最も初期の研究は State と Abadi による [9] であろう. 彼らは JVM のサブルーチンが持つ多相的な性質を分析するために型システムを定義し, サブルーチンのベリファイア上の問題点を明らかにした. 以降, 彼らの研究を基礎に様々な型システムが提案されてきた. Freund と Mitchell は [1] によって, オブジェクトの

初期化における問題点を分析するために, State と Abadi の型システムを拡張した. 彼らはサブルーチンと未初期化オブジェクトが相互作用することによって生じる問題を明らかにし, Sun のベリファイアの実装上のバグを発見した. さらに彼らは [2] によって, クラス, インターフェース, 例外などの種々の JVM の特徴を取り込んだ型システムを構築している.

Hagiya と Tozawa の [4] および, O'Callahan の [7] はサブルーチンに対する異なる型システムを定義している. Hagiya と Tozawa は健全性の証明が簡潔な型システムを提案した. O'Callahan はサブルーチンのリターンアドレスをリターン先のコードの状態 (継続) を用いて表現できることを示し, サブルーチンの再帰呼び出しやサブルーチン内からのジャンプを可能にした.

1.4 研究の動機と手法

これらの型システムは基本的に Sun のベリファイアアルゴリズム同様データフロー解析を元に定義されており, プログラム整合性を検証するためには有効である. しかしこれらは一般のプログラミング言語における型システムとは性質が異なるシステムである. そのため, そこで研究されたきた種々の有用な概念を適用することは難しい.

この問題を分析するために, 従来の型システムについてふり返ってみる. 例えばラムダ計算の型システム Λ は次のような型判定を導くための証明システムである.

$$\Gamma \rhd M : \tau$$

Γ は型環境 と呼ばれ, ラムダ式中のそれぞれの変数の型を保持する. これは, Γ の元で M が実行されると型が τ である値を計算することを意味する. Λ はプログラムを実行することなく静的に上の型判定を導き出す. つまり, プログラムの整合性のみならずプログラムがどのような計算を行なうのかということまで実行することなく求めることができるのである. この性質により, プログラムを型システムを通して構文的に種々に分析することが可能になる.

"このような優れた性質を持つ型システムを JVM に対して適用することは可能だろうか？" というのが本研究を始めるに至った動機である。

これを実現する手がかりは論理学に求めることができる. Λ と論理学の自然演繹システム $\mathcal{N}$ の間には Curry-Howard 同型対応とよばれる対応関係が成り立つ. これは論理学における証明とラムダ計算におけるプログラムが同等の関係にあることを示すものであり, この関係によりラムダ計算の型システム Λ は $\mathcal{N}$ から導き出された. つま, もい Java バイトコードに対応するような証明システムが存在すれば, Curry-Howard 同型対応によってバイトコードに対する型システムを構築することが可能であると考えられる. 最近の研究により Java バイトコードのような低レベルなマシンコードに対応するような証明システムが発見されている. 本研究は, この証明システムを拡張することで, JVM の型システムを構築する.

1.5 研究の目的

このような考えの下, 本論文では JVM を論理学的に分析し, Curry-Howard 同型対応に基づく JVM の型システムを構築する. 我々が目標とするのは以下の４つである.

- **型システムの定義**
  型システムを構築する上で基礎となる概念は、Sequential sequent calculus と呼ばれる証明システムである。これは低レベルなマシンコードが論理学の証明システムに対応していることを明らかにした。

Sequential sequent calculus を Java バイトコードに拡張することでバイトコードの型システムを構築する。同様の考えによる JAVA バイトコードに対する型システムが [5] によって提案されているが、それはクラスや継承を含まず、また健全性の証明がされていない。本研究ではクラスや継承も含めたより大きな JVM のサブセットを対象とする。

- **型システムの健全性の証明**
  型システムの健全性とは型システムの正しさを保証するものであり, 型システムが必ず満たすべき性質である. この性質により、型システムが導出したプログラムの型が、実際にプログラムを実行して得られる値の型に一致することが保証される。バイトコードの操作的意味論を定義することで、この健全性の証明を行なう。

- **JVM の型推論アルゴリズムの構築**
  JVM のサブルーチンが持つ多相性を表現するためにプログラムを ML における多相的 let 式に類似した項とみなすことで、定義した型システムを多相型システムに拡張することができ、この型システムに対して ML の型推論アルゴリズム $\mathcal{W}$ と同様の考え方を適用することにより、プログラムの型推論アルゴリズムが構築できる。バイトコードベリファイアの問題は、型システムにおける型の導出可能性の問題に帰着できるため型推論アルゴリズムによりバイトコードベリファイアが可能になる。

- **型推論アルゴリズムの実装**
  型推論アルゴリズムを実装し, Sun のベリファイアと比較してその能力を検証する.

1.6 論文の構成

この論文は以下のように構成される. 第 2 章では JVM の内部構造について調べ, クラスファイルおよびバイトコードの特徴にちて説明する. 第 3 章では本論文が基礎とする Sequential sequent calculus の概念を説明し, JAVA バイトコードにどのように拡張できるのかについて言及する. サブルーチンが証明変換としてモデル化できることを示す. 第 4 章では JVM に対する型システムを定義する. さらに, 操作的意味論に基づきその健全性を証明して型システムの正しさを保証する. 第 5 章では, 型推論アルゴリズムを構築し, JAVA バイトコードベリファイアが型推論アルゴリズムとして記述できることを示す. 第 6 章でこの型推論アルゴリズムを実装および評価を行なう. 第 7 章でこの論文をまとめる.

第2章　JVMの構造と特徴

JVM はソフトウェアから構成される仮想的なハードウェアであり実際のマシン同様 JAVA バイトコードと呼ばれるマシン語が存在し、JVM はクラスファイルと呼ばれる JAVA バイトコードから構成される実行ファイルを読み込みそれを実行することができる. クラスファイルは一般に JAVA ソースプログラムから JAVA コンパイラによって生成され JAVA のクラスと１対１で対応している. この特徴により, JVM は JAVA が持つ大部分の特徴を受け継いでいるため一般のマシンよりも複雑な構造である。

本章では JVM の内部構造について説明し, クラスファイルおよびバイトコードの詳細について述べる.

2.1　構造

図 2.1 は JVM の一般的な構造である.

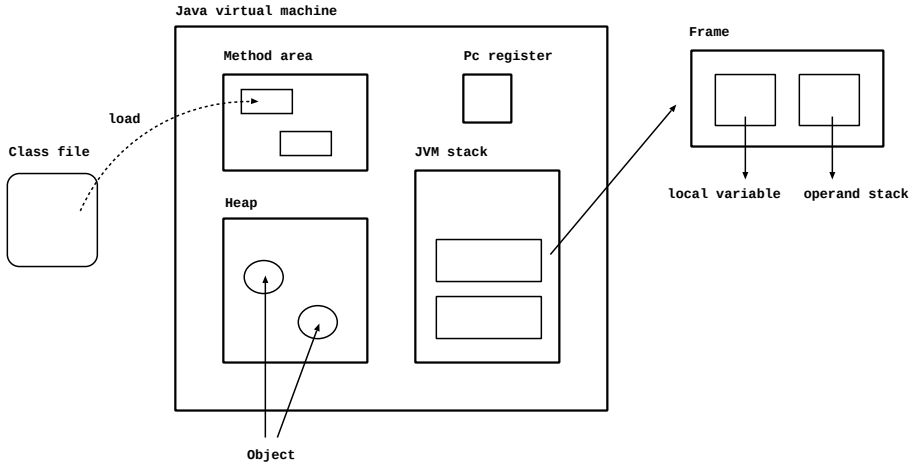

図 2.1: JVM の構造

図に示すように, JVM を構成する主な要素はメソッドエリア, ヒープ, JVM スタック, および pc レジスタであり, クラスファイルをロードすることでその実行を開始する.(ここではスレッドは考えない) 以下, それら 4 つの構造について述べる.

- **メソッドエリア**
  JVM はクラスファイルと呼ばれるバイナリ形式のファイルをロードして実行する. ロードされたクラスファイルは内部形式に変換されてメソッドエリアと呼ばれる領域に格納される. 一般の言語のコンパイル済コードの格納場所に相当する静的な領域である.

- **ヒープ**
  Java はオブジェクト指向言語であり, クラスやオブジェクトの概念を持つ. ヒープはプログラムの実

行中に生成されたインスタンスオブジェクトを格納する領域である.

- **JVM スタック**
  JVM スタックはフレームを保持する. フレームとはメソッドが実行中に使用する記憶領域でありメソッドと１対１で対応する. 実行中にメソッド呼び出しが起こると, 対応する新たなフレームが生成されて JVM スタックにプッシュされる. 生成されたフレームはメソッド終了と同時に JVM スタックから除去される. フレーム内のデータは他のメソッドからアクセスすることは出来ない. フレームはさらに
  - ローカル変数
  - オペランドスタック

  の二つのメモリ領域から構成され, メソッドはこれらの領域にアクセスしながら実行を行なう.

  ローカル変数は変数の配列であり, その大きさはコンパイル時に決定されクラスファイルに明示的に与えられる. Java のローカル変数は型が固定されそれ以外の型の値を代入することはできないが, JVM のローカル変数は任意の型の値を保持することができる. 一般にローカル変数は Java プログラムのローカル変数とは一致しない.

  オペランドスタックはメソッドが実行時に一時的に計算結果を記憶するために使用される. スタックはフレームの生成時点では空であり, 各エントリは任意の型の値を格納できる.

- **pc レジスタ**
  pc レジスタはプログラムカウンタを保持する.JVM は実行中カレントメソッドと呼ばれる単一メソッドのコードを実行している. pc レジスタはこの現在実行中の命令のアドレスが格納されている.

2.2 クラスファイル

図 2.2 に示すように JAVA コンパイラは JAVA プログラムのそれぞれのクラスから対応するクラスファイルを生成する. そのためクラスファイルは基本的にクラスと同じ構造を持ち, フィールドやメソッドなどから構成される. クラスのメソッド本体は JAVA コンパイラによってバイトコード配列に変換される.

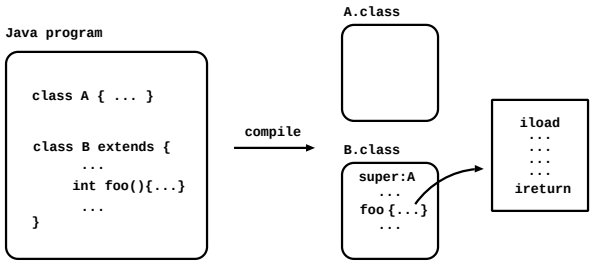

図 2.2: クラスファイル

クラスファイルは大きく分けて次の 3 つの部分に分けることができる.

- コンスタントプールテーブル
- フィールドテーブル

- メソッドテーブル

コンスタントプールテーブルはクラスファイル中で使用される全ての定数を保持する. これらの定数はコンスタントプールを通して間接的に参照することができる.

フィールドテーブルおよびメソッドテーブルはクラスにおけるフィールド定義とメソッド定義に対応する. それぞれのテーブルのエントリは名前, 型, アクセスフラグを保持し, メソッドテーブルはさらにメソッドの本体であるバイトコード配列を保持している.

クラスファイルはこの他に, 直接のスーパークラスであるクラスファイルの名前を持つ. この情報によって JVM はクラスファイル間の継承関係を知ることが可能となる.

2.3 バイトコード

Java バイトコードは JVM の命令セットである. Java コンパイラは Java プログラムの個々のメソッドをバイトコード配列にコンパイルしてクラスファイルのメソッドテーブルに格納する.

バイトコードの直接の操作対象は, フレーム内のローカル変数とオペランドスタックおよびヒープである. バイトコードは Java が持つオブジェクト指向をサポートするため一般のマシンコードよりも抽象度の高い命令が存在する. 例えば, `invoke`, `new` はそれぞれ, 単独でメソッドの呼び出しおよびインスタンスオブジェクトを生成する命令である.

また, バイトコードはバイトコードベリファイアを用意にするために各命令が明示的に型つけられているため同じ操作を行なう命令が複数存在する. 例えば, `iload`, `aload` はどちらもローカル変数から値を取り出す命令であり, 前者は整数, 後者はオブジェクト参照のみを取り出すために使用される.

Java バイトコードが持つ大きな特徴は, サブルーチンと呼ばれる内部手続きを提供していることである. サブルーチンはメソッドの内部の部分的なコード列であり, 一つのまとまった処理を行なう部分である. サブルーチンはメソッド任意の場所から呼び出すことができ, 一種の手続き呼び出しのような機能を持つ.[1] サブルーチンを実現する命令は `jsr` および `ret` である. `jsr` は一つの引数を持ち, その引数が示すサブルーチンの開始アドレスにジャンプする命令である. このとき, `jsr` はサブルーチンからの戻り先を示す自身の次の命令のアドレスをオペランドスタックにプッシュする. サブルーチンは `ret` 命令によって終了する. `ret` は引数として一つのローカル変数を受け取り, その変数に格納されたアドレスにジャンプする.

サブルーチンは一般に次のような形のコード列である.

```
astore(i)
  ...
ret(i)
```

ここで, i はローカル変数を表し, `astore` はローカル変数にリターンアドレスを格納する命令である. すなわち, サブルーチンの開始時に `jsr` によってオペランドスタックにプッシュされた戻り先のアドレスを `astore` 命令によってローカル変数に保存し, `ret` 命令によってそのローカル変数のアドレスにジャンプして終了する. なぜこのような非対称な構造をしているかについては, [10, ?] を参照されたい。`ret` の戻り先はプログラムの実行時にしか分からないため, 後でみるように我々が型システムを定義する上でも種々の工夫を要する.

[1] サブルーチン呼び出しはメソッドに似た働きをするが, あくまでもメソッド内での機能であり, メソッド呼び出しのようにフレームが生成されることはない.

第3章　JVMの論理学的構造

Java バイトコードは JVM 上で実行される一種のマシンコードである. [8] はこのようなマシンコードが論理学におけるシーケント計算によく似た証明システムによってモデル化できることを示した. この証明システムを Sequential sequent calculus(SSC) と呼ぶ. かれらはマシンコードと SSC の間に Curry-Howard 同型対応が成り立つことを証明し, この対応関係から LAM と呼ばれる仮想マシンに対する型システムを定義した. この型システムはラムダ計算の型システム同様, コード列であるプログラムを項とみなしてその型を静的に検証することが可能である. これは, 本論文が JVM に対して試みようとしていることである. もし, SSC の概念を Java バイトコードに拡張することが可能ならば LAM 同様の型システムを JVM に対して構築することが可能であると考えられる. しかしながら, JAVA バイトコードは一般のマシンコードよりも複雑な構造をしているため, これを容易に実現することはできない. 最も困難な点はサブルーチンの存在である. 本章では JVM の型システムを定義するうえで基礎となる SSC を紹介し, その概念を JVM に対して適用することを試みる.

3.1　表記

本論文の以降で使用する記法を定義する. S を任意の要素からなる列とし, e をその要素とする. このとき, $e{\cdot}S$ は S の先頭に e を加えて得られる列とする. また $S\{n \mapsto e\}$ は列 S の先頭から n 番目の要素を e で置換して得られる列とする. これを拡張して, $S\{i \mapsto e_i, \cdots, i+n \mapsto e_{i+n}\}$ は列 S の i から $i+n$ の要素をそれぞれ e_i から e_{i+n} で置換した列を表すものとする. ϕ は空列を意味する. また, F を関数とするとき $F.x$ は $F(x)$ の略記法とする.

3.2　Sequential sequent calculus

SSC は低級言語であるマシンコードを分析するための証明システムとして定義され, 自然演繹システムなど既存の証明システムと同等の証明能力を持つことが証明されている. SSC は次のような形をした推論規則の集合からなる.

$$\frac{\Delta_2 \rhd \tau}{\Delta_1 \rhd \tau}$$

その特徴は次の 3 つが上げられる.

- 個々の推論規則は, 結論 τ に影響しない. 仮定の集合 Δ が変化するのみである.
- 前提となる条件は一つだけである. したがって, 証明はこれらの推論規則が分岐することなく連続的に合成したものである.
- 結論 τ は $\Delta \rhd \tau$ (if $\tau \in \Delta$) の形をした始式によってのみ定まる.

この SSC が持つ特徴は, 次のマシンコードの性質に対応する.

- 個々の命令は実行されるとマシンのメモリ状態を変化させる.
- プログラムは連続した命令列からなり, 最後は必ずリターン命令で終了する.
- プログラムの値は, リターン命令によってのみ定まる.

つまり, 仮定の集合 Δ をマシンのメモリ, 結論 τ をプログラムの計算結果とみなすと, 推論規則はマシンコードの命令に対応し, プログラムはこれらの推論規則から合成される証明と考えることができる. このとき, リターン命令は始式 $\Delta \rhd \tau$ に一致する.

この SSC とマシンコードの対応関係から, マシンコードに対する型システムは自然に導かれる. I を命令, C をコード列, $I{\cdot}C$ を C の先頭に I 加えて得られるコード列とすると, I は

$$\frac{\Delta_2 \rhd C : \tau}{\Delta_1 \rhd I{\cdot}C : \tau}$$

の型付け規則によってモデル化できる. これはコード列 $I{\cdot}C$ を実行した結果の型が τ であり, I の実行によってメモリ (ここではスタックとする) の状態が Δ_1 から Δ_2 に変化したことを表現する. マシンコードの型システムは個々の命令に対応するこのような型付け規則から構成される. 例えば, スタックの先頭の値を取り除く命令 pop の型付け規則は次のようになる.

$$\frac{\Delta \rhd C : \tau}{\tau'{\cdot}\Delta \rhd \mathtt{pop}{\cdot}C : \tau}$$

return 命令は型システムにおける始式であり, $\tau{\cdot}\Delta \rhd \mathtt{return} : \tau$ と解釈することができる. プログラム (コード列) の型判定はこの型付け規則の合成によって生成される次のような証明によって与えられる.

$$\cfrac{\cfrac{\cfrac{}{\Delta_n \rhd \mathtt{return} : \tau}}{\cdots}}{\Delta_2 \rhd I_2{\cdot}\cdots{\cdot}\mathtt{return} : \tau}}{\Delta_1 \rhd I_1{\cdot}I_2{\cdot}\cdots{\cdot}\mathtt{return} : \tau}$$

これは, コード列 $I_1{\cdot}I_2{\cdot}\cdots{\cdot}\mathtt{return}$ から成るプログラムはメモリ状態が Δ_1 から実行を開始すると, τ である値を計算して返すことを意味する. ここで, プログラムの実行は証明の最後の推論規則を取り除くことに対応し, 証明の導出方向とは逆であることに注意する.

3.3 バイトコードの論理的解釈

JVM はメソッドのバイトコード配列を実行する. 個々のメソッドは独立し, それぞれのメソッドは単体で実行されるためメソッドのバイトコード配列が JVM におけるプログラムの最小単位であるとみなすことができる. 以降メソッドのバイトコード配列をプログラムと呼ぶ. Java バイトコードにはジャンプ命令が存在するためプログラムは単なる連続的な命令列ではない. それらはラベル付けされたいくつかのブロックから構成されると考えるのが自然である. それぞれのブロック内部は連続した命令列でありその最後は必ずジャンプ命令あるいはリターン命令によって終了する. ブロックを構成する個々のバイトコードはフレーム内のローカル変数およびオペランドスタックを操作する. 以上の考えから, SSC の概念に従うとブロック B は次のような型判定を持つ証明とみなすことができ,

$$\Gamma, \Delta \rhd B : \tau$$

個々の命令は次のような型付け規則によってモデル化される.

$$\frac{\Gamma_2, \Delta_2 \rhd B : \tau}{\Gamma_1, \Delta_1 \rhd I{\cdot}B : \tau}$$

ここで, Γ はローカル変数の型を保持するローカル環境, Δ はオペランドスタック内の型を保持するスタック環境である. 前節でみたように, リターン命令はそれ自身がプログラムであり型システムにおける公理とみなせる. 例えば ireturn はオペランドスタックのトップの整数をリターンする命令であるので, $\Gamma, \mathsf{int}{\cdot}\Delta \rhd \texttt{ireturn} : \mathsf{int}$ と表現することができる. ジャンプ命令は既に存在する証明を参照する公理とみなすことができる. 例えば, goto(l) は l でラベル付けされたブロック B にジャンプする命令であり,

$$\Gamma, \Delta \rhd \texttt{goto}(l) : \tau \quad (\text{if } \Gamma, \Delta \rhd B : \tau)$$

と解釈することができる. Java バイトコードの大部分の命令はこの考え方に従ってモデル化することが可能である. ところが前章で示したように, ジャンプ命令にはサブルーチンを呼び出す jsr とサブルーチンからリターンする ret が存在する. これらの命令は上で示した単純な型付け規則でモデル化することはできない. 例えば ret(i) はローカル変数 i に格納されたアドレスにジャンプ (リターン) する命令であり, i が示すブロックを B とすると

$$\Gamma, \Delta \rhd \texttt{ret}(x) : \tau \quad (\text{if } \Gamma, \Delta \rhd B : \tau)$$

と表すことができるが, i の内容は実行時にしか分からないためこのような型付け規則は意味をなさない.

この問題を解決するためサブルーチンを一般のブロックとは異なるサブルーチンブロックと考え, 次のように解釈する.

- サブルーチンはブロックから呼び出される, 与えられたメモリ状態を別のメモリ状態に変化させる一種の関数である.

この解釈の下では, サブルーチンは与えられた証明を拡張する証明変換とみなすことができる. 例えば今, 図.3.1 に示すように, SB を l でラベル付けされたサブルーチン, B を $\Gamma_1, \Delta_1 \rhd B : \tau$ なるブロック, $SB@B$ を B のトップに SB のコード列を付加して得られるブロックとする. このとき, $\Gamma_2, \Delta_2 \rhd SB@B : \tau$ が得られるとき, SB は証明 $\Gamma_1, \Delta_1 \rhd \tau$ を $\Gamma_2, \Delta_2 \rhd \tau$ に変換する証明変換と解釈することができる. 以上の考えからサブルーチンは次のような型判定を持つものとする.

$$SB : \langle \Gamma_1, \Delta_1 \rhd \tau // \Gamma_2, \Delta_2 \rhd \tau \rangle$$

ret(i) はサブルーチンからリターンする命令であり, ローカル変数の i 番目の要素に格納されたリターンアドレスにジャンプする. 図 3.1 において, リターンアドレスは jsr の次の命令, すなわちブロック B である. つまり, 型理論的な立場からするとリターンアドレスの型は B が持つ型, すなわち $\Gamma_1, \Delta_1 \rhd \tau$ とみなすことができる. このとき, B にリターン直後はローカル変数の i 番目の要素にはリターンアドレスが含まれているはずであるから次のような等式が成りたつ.

$$\Gamma_1(i) = \Gamma_1, \Delta_1 \rhd \tau$$

これはリターンアドレスの型が次のような再帰的な型として表現できることを意味している.

$$\alpha_l = \Gamma_1, \Delta_1 \rhd \tau$$

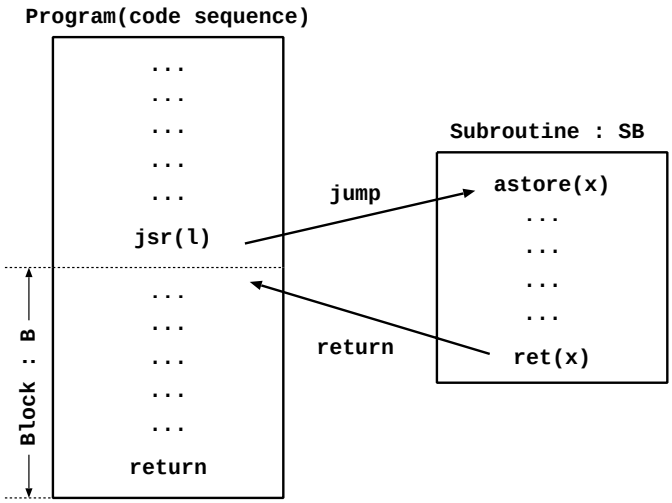

図 3.1: サブルーチンの呼び出し

ここで, α_l は型変数であり, l でラベル付けされたサブルーチンのリターンアドレスを表す型変数である. この定義のもと, サブルーチンの型付けは次のように変更される.

$$(\alpha_l = \Gamma_1, \Delta_1 \rhd \tau \text{ in } \langle \alpha_l // \Gamma_2, \Delta_2 \rhd \tau \rangle)$$

ここで、等式 $\alpha_l = \Gamma_1, \Delta_1 \rhd \tau$ はグローバルな宣言として与えられる。

第4章　JVMに対する型システムの構築

前章でJVMのバイトコードがシーケント計算に似た証明システムとしてモデル化できることを述べた. そこでカギとなる概念はサブルーチンを証明変換を行なう関数とみなすことである. 本章では前章で示された考えを基礎にJVMのサブセットに対する型システムを構築し, 操作的意味論にもとづいてその健全性の証明を行なう. このサブセットはクラスや継承などの概念を含んだ十分に大きなものであり JVM-と呼ぶことにする.

4.1　JVM-の特徴

JVM-はJVMのサブセットであり, 次の性質を持つ.

- クラス, 継承, オブジェクト
- インスタンスメソッド呼び出し
- サブルーチン
- 整数型およびオブジェクト参照型

この論文の目的は, 従来提案されてきた型システムとは本質的に異なる型システムを提案することであり, これらの特徴はJVMを分析するためには十分な大きさを持つものと考えられる. その他のJVMの特徴, 例えばスレッド, インターフェース, 例外, 静的属性, メソッドのオーバーロードなどについては将来の研究とする.

4.2　JVM-の構文規則

JVM同様 JVM-もプログラムはクラスファイルの集合からなる. 個々のクラスファイルはJAVAプログラムのクラスに対応し, フィールドやメソッドから構成される. それぞれは名前および型から構成され, さらにメソッドはバイトコード配列を保持する. ここで, 型とコードを明確に区別するため, クラスファイルの集合ををその型情報を定義する部分 Θ とメソッド本体を定義する部分 Π の組 (Θ, Π) に抽象化する. 前章で, クラスファイルにはそのクラスのスーパークラスが明示的に与えられていることは述べた. この情報からJVMはクラス間の継承関係を知ることができる. 以降ではクラス間の継承関係は暗黙的に与えられているものと仮定し, クラス c_1 が c_2 のサブクラスであることを $c_1 <: c_2$ と書く.

Θ はクラス名 c からクラス仕様 $spec$ への関数であり, それぞれのクラス仕様は型付のフィールド名 f および型付のメソッド名 m の集合である.Θ の定義を下に示す.

$$\begin{aligned} \Theta &:= \{c = spec, \ldots, c = sepc\} \\ spec &:= \{\mathtt{methods} = \{m : \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau, \cdots, m : \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau\}, \end{aligned}$$

$$\texttt{fields} = \{f : \tau, \cdots, f : \tau\}\}$$
$$\tau \quad := \quad \mathsf{int} \mid \mathsf{void} \mid c \mid \alpha_l \mid \top$$

ここで τ は型を表す. 型 c はオブジェクト型を意味し, そのオブジェクトのクラス名を型として用いる. 型 $\top$ は未定義の型を表し, この型を持つ変数にはアクセス出来きないことを意味する.$\{\tau_1, \ldots, \tau_n\} \Rightarrow \tau$ はメソッドの型であり, $\{\tau_1, \ldots, \tau_n\}$ がそれぞれの引数の型, τ が返り値の型を示す. メソッド呼び出しの際に引数として渡される自身のオブジェクトへの参照は明示的に引数には含めていない.

Π はクラス名からそのクラスが定義するメソッド本体への関数として与えられる. JVM ではメソッド本体は単一のバイトコード列であるが, ジャンプ命令が存在するため, 前章でみたようにラベル付けされたサブルーチンを含むいくつかのコードブロックに分けることができる. 図.4.1 に示すように, 一般にサブルーチンはさらに小さなコードブロックの集合からなる. JVM の仕様ではサブルーチンは `jsr` によってのみ呼

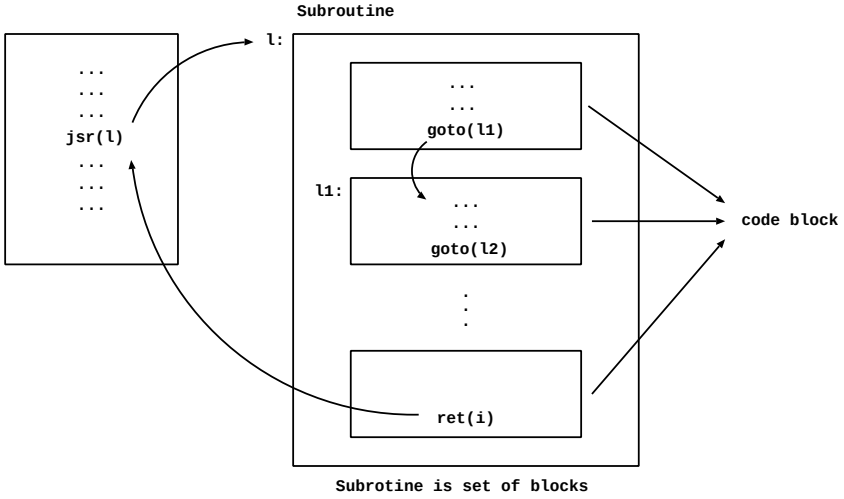

図 4.1: サブルーチンの構造

び出され, 復帰を行なう `ret` は複数のサブルーチンに属することは出来ない. したがって, サブルーチンを構成するこれらのブロックは必ず唯一のサブルーチンに属するはずである. これらサブルーチンを構成するコードブロックをサブルーチンブロックと呼び SB で表し, $SB(l)$ はサブルーチンブロック SB が l でラベル付けされたサブルーチンに所属することを表す. また, サブルーチン以外のコードブロックをベーシックブロックと呼び B で表す. 以上の考えの下 Π の定義を下に示す.

$$\begin{aligned}
\Pi &:= \{c = methods, \ldots, c = methods\} \\
methods &:= \{m = M, \ldots, m = M\} \\
M &:= \{l_b : B, \cdots, l_b : B \mid l_s : SB(l_s), \cdots, l_s : SB(l_s)\} \\
B &:= \texttt{return} \mid \texttt{ireturn} \mid \texttt{areturn} \mid \texttt{goto}(l_b) \mid \texttt{jsr}(l_s,\ l_b) \mid I{\cdot}B \\
SB &:= \texttt{ret}(i) \mid \texttt{return} \mid \texttt{ireturn} \mid \texttt{areturn} \mid \texttt{goto}(l) \mid \texttt{jsr}(l_s,\ l_s) \mid I{\cdot}SB \\
I &:= \texttt{iconst}(n) \mid \texttt{iload}(i) \mid \texttt{aload}(i) \mid \texttt{istore}(i) \mid \texttt{astore}(i) \\
&\quad \mid \texttt{dup} \mid \texttt{iadd} \mid \texttt{pop} \mid \texttt{new}(c) \mid \texttt{getfield}(c,\ f) \mid \texttt{putfield}(c,\ f) \\
&\quad \mid \texttt{invoke}(c,\ m) \mid \texttt{ifeq}(l)
\end{aligned}$$

I は JVM-の命令セットであり, その引数となる i, n, l_b, l_s はそれぞれローカル変数のインデクス (0 から始まる整数である), 整数値, ベーシックブロックラベル, サブルーチンブロックラベルを表す.JVM では jsr は引数としてサブルーチンラベルのみを持つが, JVM-では型システムの構築を容易にするため, 明示的に戻りラベルを加えている. したがって, JVM における $\mathtt{jsr}(l)B\cdot$ なるコードは $\mathtt{jsr}(l, l')$ となり, $l' : B$ がブロック集合に加えられる.

JVM-は JVM のサブセットであり、次の 3 点を仮定する。

- new は完全に初期化されたオブジェクトを生成する
  JVM では, オブジェクトの生成は次の 2 段階で行なわれる.
  - 未初期化のオブジェクトの生成. このオブジェクトにアクセスすることはできない.
  - コンストラクタメソッドでオブジェクトを初期化する.

  このオブジェクトの生成方法がバイトコードベリファイア上の微妙な問題を引き起こすが, [1, 5] で示された手法により解決できる。

- static メソッドは存在しない
  JVM-では static なメソッドは存在しないので, invoke はインスタンスメソッド呼び出しのみを行なう. またメソッドのオーバーロード機構はないため引数に型情報を必要としない.

- フィールドの隠蔽機構はない
  JVM ではスーパークラスで定義されたフィールドと同じ名前のフィールドをサブクラスで定義可能であり、このときサブクラスには両方のフィールドが存在する。JVM-ではクラス名は全て異なるものと仮定する。

4.3 型システム

先に述べたように, 型システムはプログラムの整合性を検証するものであり, それらは型システムによって導出される型判定によって保証される. このような型判定は JVM-では次の 3 つのレベルが考えられる.

1. B および SB の型判定 — ブロックおよびサブルーチンが正しい型を持つか？
2. M の型判定 — メソッドが正しい型を持つか？
3. (Θ, Π) の型判定 — プログラム定義 Π がプログラム仕様 Θ を満たすか?つまりプログラムが正しいか?

下位の型判定はより上位の型判定を用いて定義される.

4.3.1 ブロックの型付け規則

2 章で述べたようにブロック B およびサブルーチン SB の型判定は次の形を持つ.

$$\Gamma, \Delta \rhd B : \tau$$
$$SB : (\alpha_l = \Gamma, \Delta \rhd \tau \text{ in } \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle)$$

これらのブロックにはジャンプ命令が存在し, ラベルによって他のブロックを参照する. したがって, 型判定を行なうためには全てのブロックの型を保持する環境が必要である. このため次の形をしたラベル環境 $\mathcal{L}$ を導入する.

$$\mathcal{L} = \{l_b : \Gamma, \Delta \rhd \tau, \cdots \mid l_s : (\alpha_l = \Gamma_1, \Delta_1 \rhd \tau \text{ in } \langle \alpha_l // \Gamma_2, \Delta_2 \rhd \tau, \cdots \rangle)\}$$

このラベル環境 $\mathcal{L}$ の元, B および SB の型判定は次のように表記する.

$$\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau$$
$$\mathcal{L} \vdash SB : (\alpha_l = \Gamma_1, \Delta_1 \rhd \tau \text{ in } \langle \alpha_l // \Gamma_2, \Delta_2 \rhd \tau \rangle)$$

これは, " ラベル環境 $\mathcal{L}$ の下, ブロック B は型 $\Gamma, \Delta \rhd \tau$ を持ち, サブルーチンブロック SB は型 $(\alpha_l = \Gamma_1, \Delta_1 \rhd \tau \text{ in } \langle \alpha_l // \Gamma_2, \Delta_2 \rhd \tau, \cdots \rangle)$ を持つ" と読むこの型判定を導出するための型付け規則を図 4.2 および図 4.3 に示す. 図 4.3 はサブルーチンの型付け規則である. ここで, リターンアドレスの型が無視できるときは $\langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle$ と表記する.

4.3.2 メソッドの型付け規則

メソッドはブロックの集合である. メソッドが正しい型を持つのは, それを構成する全てのブロックが正しい型を持つときである. これは次の型判定によって定義される.

$$\begin{aligned} \vdash M : \mathcal{L} \Longleftrightarrow \quad & \forall l \in dom(M). \mathcal{L}(l_b) = \Gamma, \Delta \rhd \tau \wedge \mathcal{L} \vdash \Gamma, \Delta \rhd M(l_b) : \tau \\ & \text{and } \forall l_s \in dom(M). \mathcal{L} \vdash M(l_s) : \mathcal{L}(l_s) \end{aligned}$$

これはメソッドの全てのブロックがラベル環境の対応するラベルと同じ型を持つことを意味し, メソッド M がラベル環境 $\mathcal{L}$ を満たすという. この型判定はメソッドの各ブロックが正しい型を持つことを示しただけであり, メソッドが持つ型を判定することはできない. さて, メソッドを構成するブロックには必ず一つエントリブロックが存在する. エントリブロックとはメソッドが呼び出されたとき, 最初に実行されるブロックである. ラベル集合の中にこのエントリラベルが唯一存在し, これを $entry$ と表すことにすると, 次の型判定が得られる.

$$\begin{aligned} & \vdash M : \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau \\ & \Longleftrightarrow \exists \mathcal{L}. \vdash M : \mathcal{L} \wedge \mathcal{L} \vdash Top(max_{entry})\{0 \mapsto c, 1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}, \phi \rhd M.entry : \tau \end{aligned}$$

ここで, $Top(n)$ は大きさが n の全ての値が $\top$ のローカル変数環境を作成する関数であり, max_{entry} はメソッドが使用するローカル変数の大きさであり, これは JVM ではクラスファイルに明示的に示されている. この型判定規則は, M が引数 $\{\tau_1, \cdots, \tau_n\}$ によって呼び出されたとき型 τ の値を返すことを意味する. 引数には暗黙的にオブジェクト型が含まれていることに注意しなければならない.

4.3.3 プログラムの型付け規則

以上の型判定によって, JVM-のプログラムの正しさは次のように定義できる.

$$\begin{aligned} & \vdash \Pi : \Theta \\ & \Longleftrightarrow dom(\Pi) = dom(\Theta) \wedge \forall c \in dom(\Pi). \forall m \in dom(c). \vdash \Pi.c.m : \Theta.c.\texttt{methods}.m \end{aligned}$$

$$\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{return} : \mathsf{void} \qquad \mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\Delta \rhd \texttt{ireturn} : \mathsf{int} \qquad \mathcal{L} \vdash \Gamma, c{\cdot}\Delta \rhd \texttt{ireturn} : c$$

$$\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{goto}(l) : \tau \quad (\text{if } \mathcal{L}(l) = \Gamma, \Delta \rhd \tau)$$

$$\mathcal{L} \vdash \Gamma_1, \Delta_1 \rhd \texttt{jsr}(l_1, l_2) : \tau$$
$$(\text{if } \mathcal{L}(l_1) = (\alpha_{l_1} = \Gamma_2, \Delta_2 \rhd \tau \text{ in } \langle \alpha_{l_1} // \Gamma_1, \alpha_{l_1}{\cdot}\Delta_1 \rhd \tau \rangle) \wedge \mathcal{L}(l_2) = \Gamma_2, \Delta_2 \rhd \tau)$$

$$\frac{\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\Delta \rhd \texttt{ifeq}(l){\cdot}B : \tau} (\text{if } \mathcal{L}(l) = \Gamma, \Delta \rhd \tau) \qquad \frac{\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{iconst}(n){\cdot}B : \tau}{\mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\Delta \rhd B : \tau}$$

$$\frac{\mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\Delta \rhd B : \tau \quad \Gamma(n) = \mathsf{int}}{\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{iload}(n){\cdot}B : \tau} \qquad \frac{\mathcal{L} \vdash \Gamma, c{\cdot}\Delta \rhd B : \tau \quad \Gamma(n) = c}{\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{aload}(i){\cdot}B : \tau}$$

$$\frac{\mathcal{L} \vdash \Gamma\{i : \mathsf{int}\}, \Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\Delta \rhd \texttt{istore}(i){\cdot}B : \tau} \qquad \frac{\mathcal{L} \vdash \Gamma\{i : c\}, \Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, c{\cdot}\Delta \rhd \texttt{astore}(i){\cdot}B : \tau}$$

$$\frac{\mathcal{L} \vdash \Gamma\{i : \alpha_l\}, \Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \alpha_l{\cdot}\Delta \rhd \texttt{astore}(i){\cdot}B : \tau} \qquad \frac{\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \tau'{\cdot}\Delta \rhd \texttt{pop}{\cdot}B : \tau} \qquad \frac{\mathcal{L} \vdash \Gamma, \tau'{\cdot}\tau'{\cdot}\Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \tau'{\cdot}\Delta \rhd \texttt{dup}{\cdot}B : \tau}$$

$$\frac{\mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\mathsf{int}{\cdot}\Delta \rhd \texttt{iadd}{\cdot}B : \tau} \qquad \frac{\mathcal{L} \vdash \Gamma, c{\cdot}\Delta \rhd B : \tau \quad c \in Dom(\Theta)}{\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{new}(c){\cdot}B : \tau}$$

$$\frac{\mathcal{L} \vdash \Gamma, \Theta.c.\texttt{fields}.f{\cdot}\Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, c_0{\cdot}\Delta \rhd \texttt{getfield}(c, f){\cdot}B : \tau} (\text{if } c_0 <: c)$$

$$\frac{\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \tau_0{\cdot}c_0{\cdot}\Delta \rhd \texttt{putfield}(c, f){\cdot}B : \tau} (\text{if } c_0 <: c \wedge \tau_0 <: \Theta.c.\texttt{fields}.f)$$

$$\frac{\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau' \quad \Theta.c.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Rightarrow \mathsf{void}}{\mathcal{L} \vdash \Gamma, \tau_n{\cdot}\cdots{\cdot}\tau_1{\cdot}c_0{\cdot}\Delta \rhd \texttt{invoke}(c, m){\cdot}B : \tau'}$$
$$(\text{if } c_0 <: c \wedge \tau_i <: \tau'_i \quad (1 \le i \le n))$$

$$\frac{\mathcal{L} \vdash \Gamma, \tau{\cdot}\Delta \rhd B : \tau' \quad \Theta.c.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Rightarrow \tau \wedge \tau \neq \mathsf{void}}{\mathcal{L} \vdash \Gamma, \tau_n{\cdot}\cdots{\cdot}\tau_1{\cdot}c_0{\cdot}\Delta \rhd \texttt{invoke}(c, m){\cdot}B : \tau'}$$
$$(\text{if } c_0 <: c \wedge \tau_i <: \tau'_i \quad (1 \le i \le n))$$

図 4.2: ベーシックブロックの型付け規則

これは Π 中のクラスの全てのメソッドが, クラス仕様 Θ で定義されたメソッド型を持つことを意味する.

4.4 操作的意味論

型システムの健全性を証明するため, 操作的意味論を定義する. 操作的意味論とは, プログラミング言語の式の意味を実際のマシンの状態変化によって定めることである. JVM-の操作的意味論を定義するためには, JVM-のマシン状態を定義する必要がある.

4.4.1 JVM-におけるマシン状態

マシンの状態とはすなわち実行時のメモリの状態のことである. 2 章でみたように JVM のメモリ領域 (実行時領域) にはメソッドエリア, JVM スタック, ヒープ, pc レジスタが存在する. このうちメソッドエリアは

$$\mathcal{L} \vdash \texttt{ret}(i) : (\alpha_l = \Gamma, \Delta \rhd \tau \text{ in } \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle) \quad (\text{if } \Gamma(l) = \alpha_l)$$

$$\mathcal{L} \vdash \texttt{return} : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle \qquad \mathcal{L} \vdash \texttt{ireturn} : \langle \alpha_l // \Gamma, \mathsf{int} \cdot \Delta \rhd \tau \rangle \qquad \mathcal{L} \vdash \texttt{areturn} : \langle \alpha_l // \Gamma, c \cdot \Delta \rhd \tau \rangle$$

$$\mathcal{L} \vdash \texttt{goto}(l) : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle \quad (\text{if } \mathcal{L}(l) = \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle)$$

$$\mathcal{L} \vdash \texttt{jsr}(l_1, l_2) : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle \quad (\text{if } \mathcal{L}(l_1) = (\alpha_{l_1} = \Gamma', \Delta' \rhd \tau \text{ in } \langle \alpha_{l_1} // \, \rangle) \wedge \mathcal{L}(l_2) = \langle \alpha_l // \Gamma', \Delta' \rhd \tau \rangle)$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \mathsf{int} \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{iconst}(n) \cdot SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle} \qquad \frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \mathsf{int} \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{iload}(i) \cdot SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle} \quad (\text{if } \Gamma(i) = \mathsf{int})$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, c \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{aload}(i) \cdot SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle} \quad (\text{if } \Gamma(i) = c) \qquad \frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma\{i : \mathsf{int}\}, \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{istore}(i) \cdot SB : \langle \alpha_l // \Gamma, \mathsf{int} \cdot \Delta \rhd \tau \rangle}$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma\{i : c\}, \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{astore}(i) \cdot SB : \langle \alpha_l // \Gamma, c \cdot \Delta \rhd \tau \rangle} \qquad \frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma\{i : \alpha_l\}, \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{astore}(i) \cdot SB : \langle \alpha_l // \Gamma, \alpha_l \cdot \Delta \rhd \tau \rangle}$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \tau' \cdot \tau' \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{dup} \cdot SB : \langle \alpha_l // \Gamma, \tau' \cdot \Delta \rhd \tau \rangle} \qquad \frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \mathsf{int} \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{iadd} \cdot SB : \langle \alpha_l // \Gamma, \mathsf{intint} \cdot \Delta \cdot \tau \rhd \rangle}$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{pop} \cdot SB : \langle \alpha_l // \Gamma, \tau' \cdot \Delta \rhd \tau \rangle} \qquad \frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, c \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{new}(c) \cdot SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle} \quad (\text{if } c \in Dom(\Theta))$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \tau' \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{getfield}(c_0, f) \cdot SB : \langle \alpha_l // \Gamma, c_1 \cdot \Delta \rhd \tau \rangle} \quad (\text{if } \Theta.c_0.\texttt{fields}.f = \tau' \wedge c_1 <: c_0)$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{putfield}(c_0, f) \cdot SB : \langle \alpha_l // \Gamma, \tau' \cdot c_1 \cdot \Delta \rhd \tau \rangle} \quad (\text{if } \Theta.c_0.\texttt{fields}.f = \tau' \wedge c_1 <: c_0)$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \tau_0 \cdot \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{invoke}(c_0, m) \cdot SB : \langle \alpha_l // \Gamma, \tau_n \cdot \cdots \cdot \tau_1 \cdot c_1 \cdot \Delta \rhd \tau \rangle}$$
$$(\text{if } \Theta.c_0.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Rightarrow \tau_0 \wedge \tau_0 \neq \mathsf{void} \text{ and } c_1 <: c_0 \wedge \tau_i <: \tau'_i \text{ for all } 1 \leq i \leq n)$$

$$\frac{\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle}{\mathcal{L} \vdash \texttt{invoke}(c_0, m) \cdot SB : \langle \alpha_l // \Gamma, \tau_n \cdot \cdots \cdot \tau_1 \cdot c_1 \cdot \Delta \rhd \tau \rangle}$$
$$(\text{if } \Theta.c_0.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Rightarrow \mathsf{void} \text{ and } c_1 <: c_0 \wedge \tau_i <: \tau'_i \text{ for all } 1 \leq i \leq n)$$

図 4.3: サブルーチンブロックの型付け規則

基本的に静的な領域であるため, JVM の状態は JVM スタック, ヒープ, pc レジスタの 3 つによって定まると考えることができる. JVM スタックはフレームを保持し, フレームはさらにローカル変数およびオペランドスタックから構成される. S をオペランドスタック, E をローカル変数とすると, フレームは (S, E) と表すことができる. JVM スタックの先頭のフレームはカレントフレームと呼ばれ, 現在実行中のメソッドに対応しメソッドのコード配列を実行する. つまりフレームは個々メソッドのコード配列を保持していると考えるのが自然である. よってコード列を C とするとフレームは, (S, E, C) と表現できる.pc レジスタはコードの命令アドレスを保持しフレームは pc レジスタが示すコードを実行する. もし, ジャンプ命令が存在しなければフレームはコード列を先頭から順番に実行していけばよいので pc レジスタを参照する必要はなくなるはずである. ところで JVM-ではメソッドは個々のコードブロックの集合であり, それらのコードブロックは最後まで連続的に実行することができる. 以上の考えから JVM-のフレームは次のように定義できる.

$$(S, E, M\{C\})$$

ここで C はベーシックブロック B あるいはサブルーチンブロック SB である. $M\{C\}$ はメソッド M 中のブロック C の先頭のコードを実行していることを表す. JVM スタックはフレームを保持するスタックであ

り, フレームが空になると JVM の実行は終了するため, J を JVM スタックとすると

$$\begin{aligned} J &::= \phi \\ &::= (S, E, M\{C\})\cdot J \end{aligned}$$

のように自然に表現できる. このとき, ヒープを無視すると JVM-のマシン状態は J で表される. しかし, このままではマシンの終了時にマシンの状態そのものが ϕ となってしまい, 健全性の証明が複雑になってしまう. ここで, JVM スタックはマシン状態の保存に使われ SECD マシンのダンプに相当する. つまりダンプ D を

$$\begin{aligned} D &::= \phi \\ &::= (S, E, M\{C\})\cdot D \end{aligned}$$

と定義するとマシン状態は SECD マシンと同様に $(S, E, M\{C\}, D)$ と表現できる. 以上の議論からヒープを h とすると JVM-のマシン状態は $(S, E, M\{C\}, D), h$ と与えられる.

4.4.2 実行時の値

ローカル変数 E およびスタック S は実行時の値 v を保持する列とする. ヒープ h はヒープアドレス r からオブジェクトへの関数として与えられ, クラス c のオブジェクトは $\langle f_1 = v_1, \ldots, f_n = v_n \rangle_c$ と表現する. ここで, $f_1 \cdots f_n$, $v_1 \cdots v_n$ はそれぞれフィールドおよびその値を表す. オブジェクトのフィールドはそのスーパークラスのフィールドを全て含む.JVM-ではフィールドの隠蔽機構はないため, フィールド名は全て異なることに注意する. JVM-の実行時値 v は整数値 p, オブジェクト参照値 r, リターンアドレス値 $adrs(l)$ のみである. `retAdrs(`l`)` の l はサブルーチンの戻り先ラベルを示す. JVM-では `new` はデフォルトの値でフィールドの値が初期化されたオブジェクトを生成するものとし, このデフォルト値を $\perp_\tau$ と表す. これは型 τ のデフォルト値である.

4.4.3 遷移規則

操作的意味論は一般的にマシンの状態遷移規則の集合によって定義される. I をバイトコードとすると $(S, E, M\{I\cdot C\}, D), h$ は, 現在実行中のメソッドが M でブロック C の先頭のコードである I の実行前のマシン状態を表す.I の実行後, マシン状態が $(S', E', M\{C\}, D'), h'$ に変化するとき,

$$(S, E, M\{I\cdot C\}, D), h \rightarrow (S', E', M\{C\}, D'), h'$$

と表し, 命令 I の状態遷移規則と呼ぶ. 図 4.4 に状態遷移規則の集合を示す.

ここで, これらの規則は与えられた Θ の下で定義されていることに注意する. これは JVM がメソッドエリアにロードされたクラスファイルを参照することに対応している. $TopEnv(n)$ は大きさが n の全ての要素が値 $\perp_\top$ であるローカル変数を意味する. $\perp_\top$ は型 $\top$ の意味のない値である.$max_{(c,m)}$ はクラス c のメソッド m が使用するローカル変数の大きさであり, これは明示的にクラスファイルで与えられている.

$(S, E, M\{\texttt{return}\}, (S_0, E_0, M_0\{C_0\})\cdot D_0), h \rightarrow (S_0, E_0, M_0\{C_0\}, D_0), h$
$(p\cdot S, E, M\{\texttt{ireturn}\}, (S_0, E_0, M_0\{C_0\})\cdot D_0), h \rightarrow (p\cdot S_0, E_0, M_0\{C_0\}, D_0), h$
$(r\cdot S, E, M\{\texttt{areturn}\}, (S_0, E_0, M\{C_0\})\cdot D_0), h \rightarrow (r\cdot S_0, E_0, M\{C_0\}, D_0), h$
$(S, E, M\{\texttt{goto}(l)\}, D), h \rightarrow (S, E, M\{M(l)\}, D), h$
$(0\cdot S, E, M\{\texttt{ifeq(l)}\cdot C\}, D), h \rightarrow (S, E, M\{M(l)\}, D), h$
$(p\cdot S, E, M\{\texttt{ifeq(l)}\cdot C\}, D), h \rightarrow (S, E, M\{C\}, D), h$
$(S, E, M\{\texttt{iconst}(n)\cdot C\}, D), h \rightarrow (n\cdot S, E, M\{C\}, D), h$
$(S, E, M\{\texttt{iload}(i)\cdot C\}, D), h \rightarrow (E(i)\cdot S, E, M\{C\}, D), h$
$(S, E, M\{\texttt{aload}(i)\cdot C\}, D), h \rightarrow (E(i)\cdot S, E, M\{C\}, D), h$
$(p\cdot S, E, M\{\texttt{istore}(i)\cdot C\}, D), h \rightarrow (S, E\{i \mapsto p\}, M\{C\}, D), h$
$(r\cdot S, E, M\{\texttt{astore}(i)\cdot C\}, D), h \rightarrow (S, E\{i \mapsto r\}, M\{C\}, D), h$
$(adrs(l)\cdot S, E, M\{\texttt{astore}(i)\cdot C\}, D), h \rightarrow (S, E\{i \mapsto adrs(l)\}, M\{C\}, D), h$
$(v\cdot S, E, M\{\texttt{dup}\cdot C\}, D), h \rightarrow (v\cdot v\cdot S, E, M\{C\}, D), h$
$(v\cdot S, E, M\{\texttt{pop}\cdot C\}, D), h \rightarrow (S, E, M\{C\}, D), h$
$(p\cdot p\cdot S, E, M\{\texttt{iadd}\cdot C\}, D), h \rightarrow (p\cdot S, E, M\{C\}, D), h$
$(S, E, M\{\texttt{new}(c)\cdot C\}, D), h \rightarrow (r\cdot S, E, M\{C\}, D), h\{r \leftarrow \langle f_1 = \bot_{\tau_1} \cdots f_n = \bot_{\tau_n}\rangle\}$
(if $\Theta.c.\texttt{fields} = \{f_1 : \tau_1, \cdots, f_n : \tau_n\}$ and $r \notin dom(h)$)
$(r\cdot S, E, M\{\texttt{getfield}(c, f)\cdot C\}, D), h \rightarrow (v\cdot S, E, M\{C\}, D), h$ (if $v = h(r).f$)
$(v\cdot r\cdot S, E, M\{\texttt{putfield}(c, f)\cdot C\}, D), h \rightarrow (S, E, M\{C\}, D), h\{r \mapsto Update(h, r, f, v)$
$(v_n\cdot\cdots\cdot v_1\cdot r\cdot S, E, M\{\texttt{invoke}(c, m)\cdot C\}, D), h \rightarrow (\phi, E', M'\{M'.entry\}, (S, E, M\{C\})\cdot D), h$
(if $E' = TopEnv(max(c, m))\{0 \mapsto r, \cdots, n \mapsto v_n\}$ and $\Theta.c.\texttt{methods}.m = \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau$)
$(S, E, M\{\texttt{ret(i)}\}, D), h \rightarrow (S, E, M\{M(E(i))\}, D), h$
$(S, E, M\{\texttt{jsr}(l_1, l_2)\}, D), h \rightarrow (l_2\cdot S, E, M\{M(l_1)\}, D), h$

図 4.4: JVM-に対する操作的意味論

4.5 型システムの健全性

前節で定義した操作的意味論をもとに型システムの健全性を証明する. 型システムはプログラムを実行することなくその型を決定するシステムである. 健全性とは, 型システムによって演繹されたプログラムの型は, プログラムを実際に実行するとその型の値を返すことを保証する性質である. 操作的意味論による健全性は一般に次のことを示すことで証明される

1. 任意の遷移規則についてマシンが正しい状態は遷移後もマシンの状態は必ず正しい
2. プログラムは正しい型の値を返して終了する

操作的意味論は型システムと無関係に定義されたものであり, 証明のためにまずマシン状態の正しさを定義する必要がある. マシン状態の正しさは次の型付け規則によって定義される.

- $\mathcal{L} \models h : H$　　ラベル環境 $\mathcal{L}$ の下, ヒープ h は型 H を持つ
- $\mathcal{L}, H \models v : \tau$　ラベル環境 $\mathcal{L}$ およびヒープ型 H の下, 実行時値 v は型 τ を持つ
- $\mathcal{L}, H \models S : \Delta$　$\mathcal{L}$ および H の下, スタック S はスタック環境 Δ を満たす
- $\mathcal{L}, H \models E : \Gamma$　$\mathcal{L}$ および H の下, ローカル変数 E はローカル環境 Γ を満たす

これらの規則は種々の実行時オブジェクトの正しさを保証する. [6] で示された H はヒープ型と呼ばれヒープアドレス r からオブジェクト型への関数である. ヒープを型付けることで, 実行自値 v の型付けがヒープのポインタを通して循環するのを防ぐことができる. 実行時値 v の型付け規則はつぎのように定義される.

$$\begin{aligned}
\mathcal{L}, H &\models p : int \\
\mathcal{L}, H &\models r : \tau \quad (\text{if } \tau <: H(r)) \\
\mathcal{L}, H &\models adrs(l) : \alpha_{l'} \quad (\text{if } \mathcal{L}(l) = \alpha_{l'} \text{ or } \mathcal{L}(l) = \langle \alpha_{l''} // \Gamma, \Delta \rhd \tau \rangle \text{ and } \alpha_{l''} = \Gamma, \Delta \rhd \tau)
\end{aligned}$$

ここで, 最後のリターンアドレスの型付け規則の条件は次のように解釈できる. 最初の条件はサブルーチンがベーシックブロックから呼び出されたときに対応する. 2番目の条件はサブルーチンがサブルーチンブロックから呼び出されたことに対応し, リターンアドレスの型は戻り先のサブルーチンブロックの開始時の型と一致することを意味する. これらの規則の下ヒープ h の型付け規則は

$$\begin{aligned}
\mathcal{L} \models h : H \iff\ & dom(h) = dom(H), \forall r \in dom(h) \text{ if } h(r) = \langle f_1 = v_1, \ldots, f_n = v_n \rangle_c \\
& \text{then } c <: H(r) \text{ and } \mathcal{L}; H \models v_i : \Theta.c.\texttt{fields}.f_i \text{ for each } i.
\end{aligned}$$

と定義できる. 最後にスタック S およびローカル変数 E の型付け規則は次のように定義できる.

$$\begin{aligned}
\mathcal{L}; H \models S : \Delta \iff\ & dom(S) = dom(\Delta) \text{ and } \mathcal{L}; H \models S.i : \Delta.i \text{ for each } i. \\
\mathcal{L}; H \models E : \Gamma \iff\ & dom(E) = dom(\Gamma) \text{ and } \mathcal{L}; H \models E.i : \Gamma.i \text{ for each } i.
\end{aligned}$$

マシン状態の型付けを定義するためには, さらにダンプ D の型付けを定義する必要がある. これは [8] で示された考えに従う. ダンプ D はフレームのスタックであり, トップのフレームは現在のカレントフレームの終了後に, カレントフレームからの返り値を受け取って実行を再開する. つまり D は値を受け取って計算残りの計算を行なう継続とみなすことができ, 次のように帰納的に正しさを定義することができる.

- $H \models \phi : \tau$ (τ は任意)
- $H \models (S, E, M\{C\}) \cdot D : \tau$
  もし $C = B$ ならば, ある $\Gamma, \Delta, \mathcal{L}, \tau'$ が存在して $\vdash M : \mathcal{L}$, $\mathcal{L}; H \models S : \Delta'$, $\mathcal{L}; H \models E : \Gamma$, $\mathcal{L} \vdash \Gamma, \Delta, M\{B\} \rhd \tau'$, かつ $H \models D : \tau'$ を満たす.
  もし $C = SB$ ならば, ある $\exists \Gamma, \Delta, \mathcal{L}, \tau'$ が存在して $\vdash M : \mathcal{L}$, $\mathcal{L}; H \models S : \Delta'$, $\mathcal{L}; H \models E : \Gamma$, $\mathcal{L} \vdash M\{SB\} : \langle \alpha_l // \Gamma, \Delta \rhd \tau' \rangle \Gamma, \Delta \rhd \tau'$ かつ $H \models D : \tau'$ を満たす.
  ここで, もし $\tau = \mathsf{void}$ ならば $\Delta' = \Delta$ であり, さもなければ $\Delta' = \tau \cdot \Delta$ である.

$H \models D : \tau$ はダンプ D が型 τ の値を受け取る正しい型のダンプであることを意味する.

以上の定義により, マシン状態 $(S, E, M\{C\}, D), h$ の正しさは次のように定義できる.

$$\begin{aligned}
H \vdash (S, E, M\{B\}, D), h \iff\ & \text{ある } \mathcal{L}, \Gamma, \Delta \text{ が存在して } \vdash M : \mathcal{L}, \mathcal{L} \models h : H, \mathcal{L}; H \models S : \Delta, \\
& \mathcal{L}; H \models E : \Gamma, \mathcal{L} \vdash \Gamma, \Delta, M\{B\} \rhd \tau, \text{ かつ } H \models D : \tau. \\
H \vdash (S, E, M\{SB\}, D), h \iff\ & \text{ある } \mathcal{L}, \Gamma, \Delta \text{ が存在して } \vdash M : \mathcal{L}, \mathcal{L} \models h : H, \mathcal{L}; H \models S : \Delta, \\
& \mathcal{L}; H \models E : \Gamma, \mathcal{L} \vdash B\{SB\} : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle \Gamma, \Delta \rhd \tau, \text{ かつ } H \models D : \tau.
\end{aligned}$$

以上の定義によって型システムの健全性は以下の定理で表すことができる. この定理により、正しく型付けられたマシン状態はリターン命令によって終了するか、あるいは別の正しいマシン状態へ遷移することが表される。つまり、正しいプログラムは途中で停止することなく、マシンが終了するときは型システムによって導出された型と同じ型の値をスタックのトップに存在することが保証される。

補題 1 もし $\mathcal{L}, H \models v : \tau$ かつ H' が H の拡張ならば, $\mathcal{L}, H' \models v : \tau$ である

証明: v に関する場合分けにより証明する.

p **の場合.** 成り立つ.

$adrs(l)$ **の場合.** 成り立つ.

r **の場合.** $\mathcal{L}, H \models r : c_1$ と仮定する. このとき, $H(r_1) <: c_1$ である. $H_2 = H\{r_2 \mapsto c_2\} \wedge r_2 \notin dom(H)$ ならば, $H_2(r_1) = H(r_1)$ より $H_2(r_1) <: c_1$ である. よって成り立つ.

定理 1 ***(型システムの健全性)*** $\vdash \Pi : \Theta$ であるようなプログラム (Π, Θ) を考える. もし, $H \vdash (S, E, M\{C\}, D), h$ ならば *(1)* C は `return`, `ireturn`, `areturn` のいずれかであり, かつ $D = \phi$ である. または *(2)* ある $S', E', M'\{C'\}, D', h'$ かつ H' が存在して, H' は H の拡張であり, また

$$(S, E, M\{C\}, D), h \longrightarrow (S', E', M'\{C'\}, D'), h',$$

かつ, $H' \vdash (S', E', M'\{C'\}, D'), h'$ が成り立つ. ここで C は B または SB である.

証明: 証明は $\vdash \Pi : \Theta$ より, ある $\mathcal{L}$ が存在して $\vdash M : \mathcal{L}$ が成り立つと仮定した下で行なう.
$C = B$ の場合を示す.$C = SB$ の場合も同様にして証明できる.
$H \vdash (S, E, M\{B\}, D), h$ と仮定すると, ある Γ, Δ が存在して $\mathcal{L} \models h : H, \mathcal{L}, H \models E : \Gamma, \mathcal{L}, H \models S : \Delta, \mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau$ かつ $H \models D : \tau$ が成り立つ. 以下, B の最初の命令の場合分けにより証明する.
$B =$ `return` **の場合.**
$D = \phi$ またはある S_1, E_1, M_1, B_2, D_1 が存在して $D = (S_1, E_1, M_1\{B_2\})\cdot D_1$ である. $D = \phi$ のときは成り立つ.$D = (S_1, E_1, M_1\{B_2\})\cdot D_1$ のとき, 遷移規則より $(S, E, M\{\texttt{return}\}, (S_1, E_1, M_1\{B_2\})\cdot D_1), h \rightarrow (p\cdot S_1, E_1, M_1\{B_1\}, D_2), h$ である. 型システムの定義より, $\tau =$ void である. D_1 の型付け定義より, ある $\Gamma_1, \Delta_1, \mathcal{L}_1, \tau_1$ が存在して, $\vdash M_1 : \mathcal{L}_1, \mathcal{L}_1, H \models S_1 : \Delta_1, \mathcal{L}_1 \vdash \Gamma_1, \Delta_1 \rhd M_1\{B_2\} : \tau_2$ かつ $H \models D_2 : \tau_1$ である. よって成り立つ.
$B =$ `ireturn` **の場合.**
$D_1 = \phi$ またはある S_1, E_1, M_1, B_2, D_1 が存在して $D_1 = (S_1, E_1, M_1\{B_2\})\cdot D_2$ である. $D_1 = \phi$ のときは成り立つ.$D_1 = (S_1, E_1, M_1\{B_2\})\cdot D_2$ のとき, 型システムの定義より, ある Δ' が存在して $\Delta =$ int$\cdot\Delta'$ かつ $\tau =$ int である. またこのとき, ある S' が存在して $S = p\cdot S', \mathcal{L}, H \models S' : \Delta'$. 遷移規則より $(p\cdot S', E, M\{\texttt{ireturn}\}, (S_1, E_1, M_1\{B_2\})\cdot D_2), h \rightarrow (p\cdot S_1, E_1, M_1\{B_1\}, D_2), h$ である. D_1 の型付け定義より, ある $\Gamma_1, \Delta_1, \mathcal{L}_1, \tau_1$ が存在して, $\vdash M_1 : \mathcal{L}_1, \mathcal{L}_1, H \models S_1 :$ int$\cdot\Delta_1, \mathcal{L}_1 \vdash \Gamma_1, \Delta_1 \rhd M_1\{B_2\} : \tau_2$ かつ $H \models D_2 : \tau_1$ である. よって成り立つ.
$B =$ `areturn` **の場合.**
$D_1 = \phi$ またはある S_1, E_1, M_1, B_2, D_1 が存在して $D_1 = (S_1, E_1, M_1\{B_2\})\cdot D_2$ である. $D_1 = \phi$ のときは成り立つ.$D_1 = (S_1, E_1, M_1\{B_2\})\cdot D_2$ のとき, 型システムの定義より, ある Δ' が存在して $\Delta = c\cdot\Delta', \tau = c_1$ である. このときある r_1, S' が存在して, $S = r_1\cdot S', \mathcal{L}, H \models r_1\cdot S' : c\cdot\Delta'$. 遷移規則より $(c\cdot S', E, M\{\texttt{ireturn}\}, (S_1, E_1, M_1\{B_2\})\cdot D_2), h \rightarrow (c\cdot S_1, E_1, M_1\{B_1\}, D_2), h$ である. D_1 の型付け定義より, ある $\Gamma_1, \Delta_1, \mathcal{L}, \tau_1$ が存在して, $\vdash M_1 : \mathcal{L}_1, \mathcal{L}_1, H \models S_1 : c\cdot\Delta_1, \mathcal{L}_1 \vdash \Gamma_1, \Delta_1 \rhd M_1\{B_2\} : \tau_2$ かつ $H \models D_2 : \tau_1$ である. よって成り立つ.
$B =$ `goto`(l) **の場合.**
遷移規則より, $(S, E, M\{\texttt{goto}(l)\}, D_1), h \rightarrow (S, E, M\{M(l)\}, D_1), h$ である. また, 型システムの定義より, $\mathcal{L}(l) = \Gamma, \Delta \rhd \tau$ であるから $\mathcal{L} \vdash \Gamma, \Delta \rhd M(l) : \tau$ である. よって成り立つ.

$B = \texttt{jsr}(l_1, l_2)$ **の場合.**
遷移規則より, $(S, E, \texttt{jsr}(l_1, l_2)\{M\}, D_1), h \to (adrs(l_2)\cdot S_1, E, M\{M(l_1)\}, D_1), h$ である. 型システムの定義より, ある Γ_1, Δ_1 が存在して $\mathcal{L}(l_1) = (\alpha_{l_1} = \Gamma_1, \Delta_1 \rhd \tau \text{ in } \langle \alpha_{l_1} // \Gamma, \alpha_{l_1}\cdot\Delta \rhd \tau\rangle)$, $\mathcal{L}(l_2) = \Gamma_1, \Delta_1 \rhd \tau$ が成り立つ. このとき, $M(l_1) : (\alpha_{l_1} = \Gamma_1, \Delta_1 \rhd \tau \text{ in } \langle \alpha_{l_1} // \Gamma, \alpha_{l_1}\cdot\Delta \rhd \tau\rangle)$ である. 従って $\mathcal{L}, H \models adrs(l_2) : \alpha_{l_1}$ を示せばよい. $\mathcal{L}l_2 = \Gamma_1, \Delta_1 \rhd \tau$ より $\mathcal{L}, H \models adrs(l_2) : \Gamma_1, \Delta_1 \rhd \tau$ であり, また $\alpha_{l_1} = \Gamma_1, \Delta_1 \rhd \tau$ であるから $\mathcal{L}, H \models adrs(l_2) : \alpha_{l_1}$ である. よって成り立つ.
$B = \texttt{ifeq}(l)\cdot B_1$ **の場合.**
型システムの定義より, ある Δ' が存在して $\Delta = \mathsf{int}\cdot\Delta'$ である. このとき, ある S' が存在して $S = 0\cdot S'$ または $S = p\cdot S', p \neq 0$ である. $S = 0\cdot S'$ のとき, 遷移規則より, $(0\cdot S', E, M\{\texttt{ifeq}(l)\cdot B_1\}, D_1), h \to (S', E, M\{M(l)\}, D_1), h$ である. である. また型システムの定義より $\mathcal{L}(l) = \Gamma, \Delta' \rhd \tau :$ であるから, $\mathcal{L} \vdash \Gamma, \Delta' \rhd M(l) : \tau$ である. よって成り立つ. $S = p\cdot S', p \neq 0$ のとき, 遷移規則より, $(0\cdot S', E, M\{\texttt{ifeq}(l)\cdot B_1\}, D_1), h \to (S', E, M\{B_1\}, D_1), h$ である. また, 型システムの定義より $\mathcal{L} \vdash \Gamma, \Delta' \rhd B_1 : \tau$ である. よって成り立つ.
$B = \texttt{iload}(i)B_1\cdot$ **の場合.**
遷移規則より, $(S, E, M\{\texttt{iload}(i)\cdot B_1\}, D_1), h \to (E(i)\cdot S, E, M\{B_1\}, D_1), h$ である. 型システムの定義より, $\mathcal{L} \vdash \Gamma, \mathsf{int}\cdot\Delta \rhd B_1 : \tau$ かつ $\Gamma(i) = \mathsf{int}$ である. このとき, $\mathcal{L}, H \models E_(i) : \Gamma(i)$ である. よって成り立つ.
$B = \texttt{aload}(i)B_1\cdot$ **の場合.**
遷移規則より, $(S, E, M\{\texttt{aload}(i)\cdot B_1\}, D_1), h \to (E(i)\cdot S, E, M\{B_1\}, D_1), h$ である. 型システムの定義より, $\mathcal{L} \vdash \Gamma, c\cdot\Delta \rhd B_1 : \tau$ かつ $\Gamma(i) = c$ である.$\mathcal{L}, H \models E_(i) : \Gamma(i)$ であるから成り立つ.
$B = \texttt{istore}(i)B_1\cdot$ **の場合.**
型システムの定義より, ある Δ' が存在して, $\Delta = \mathsf{int}\cdot\Delta'$ である. このとき, ある S' が存在して $S = p\cdot S'$ かつ $\mathcal{L}, H \models S' : \Delta'$ である. 遷移規則より, $(p\cdot S', E, M\{\texttt{istore}(i)\cdot B_1\}, D_1), h \to (S', E\{i \mapsto p\}, M\{B_1\}, D_1), h$ である. 型システムの定義より, $\mathcal{L} \vdash \Gamma\{i \mapsto \mathsf{int}\}, \Delta' \rhd \tau :$ である. $\mathcal{L}, H \models E\{i \mapsto p\} : \Gamma\{i \mapsto \mathsf{int}\}$ より成り立つ.
$B = \texttt{astore}(i)B_1\cdot$ **の場合.**
型システムの定義より, ある c_1, α_l, Δ' が存在して, $\Delta = c_1\cdot\Delta'$ または $\Delta = \alpha_l\cdot\Delta$ である. $\Delta = c_1\cdot\Delta'$ のとき, ある r_1, S' が存在して $S = r_1\cdot S'$ かつ $\mathcal{L}, H \models S' : \Delta', \mathcal{L}, H \models r_1 = c_1$ である. 遷移規則より, $(r_1\cdot S', E, M\{\texttt{istore}(i)\cdot B_1\}, D_1), h \to (S, E\{i \mapsto r_1\}, M\{B_1\}, D_1), h$ である. 型システムの定義より, $\mathcal{L} \vdash \Gamma\{i \mapsto c_1\}, \Delta' \rhd \tau :$ である. $\mathcal{L}, H \models E\{i \mapsto r_1\} : \Gamma\{i \mapsto c_1\}$ より成り立つ. $\Delta = \alpha_l\cdot\Delta'$ のとき, ある $adrs(l'), S'$ が存在して $S = adrs(l')\cdot S'$ かつ $\mathcal{L}, H \models S' : \Delta', \mathcal{L}, H \models adrs(l') = \alpha_l$ である. 遷移規則より, $(adrs(l')\cdot S', E, M\{\texttt{istore}(i)\cdot B_1\}, D_1), h \to (S, E\{i \mapsto adrs(l')\}, M\{B_1\}, D_1), h$ である. また型システムの定義より, $\mathcal{L} \vdash \Gamma\{i \mapsto \alpha_l\}, \Delta' \rhd \tau :$. $\mathcal{L}, H \models E\{i \mapsto adrs(l')\} : \Gamma\{i \mapsto \alpha_l\}$ より成り立つ.
$B = \mathsf{dup}\cdot B_1$ **の場合.**
型システムの定義より, ある τ_1, Δ' が存在して $\Delta = \tau_1\cdot\Delta'$ である. このとき, ある v_1, S' が存在して $S = v_1\cdot S', \mathcal{L}, H \models v_1\cdot S' : \tau_2\cdot\Delta'$. 遷移規則より, $(v_1\cdot S', E, M\{\mathsf{dup}\cdot B_1\}, D_1), h \to (v_1\cdot v_1\cdot S, E, M\{B_1\}, D_1), h_2$ である. 型システムの定義より, $\mathcal{L} \vdash \Gamma, \tau_1\cdot\tau_1\cdot\Delta' \rhd B_1 : \tau$. このとき, $\mathcal{L}, H \models v_1\cdot v_1\cdot S' : \tau_2\cdot\tau_2\cdot\Delta'$ であるから成り立つ.
$B = \mathsf{iadd}\cdot B_1$ **の場合.**
型システムの定義より, ある Δ' が存在して, $\Delta = \mathsf{int}\cdot\mathsf{int}\cdot\Delta'$ である. このとき, ある S' が存在して, $S = p\cdot p\cdot S', \mathcal{L}, H \models S' : \Delta'$ であある. 遷移規則より, $(p\cdot p\cdot S', E, M\{\mathsf{iadd}\cdot B_1\}, D_1), h \to (p\cdot S', E, M\{B_1\}, D_1), h_2$ である. 型システムの定義より, $\mathcal{L} \vdash \Gamma, \mathsf{int}\cdot\Delta' \rhd B_1 : \tau$. よって成り立つ.
$B = \texttt{new}(c)\cdot B_1$ **の場合.**

遷移規則より, $(S, E, M\{\texttt{new}(c)\cdot B_1\}, D), h \rightarrow (r\cdot S, E, M\{B_1\}, D), h_1$ かつ $h_1 = h\{r \mapsto \langle f_1 = \bot_{\tau_1}, \cdots, f_n = \bot_{\tau_n}\rangle_c\} \wedge \Theta.c.\texttt{fields} = \{f_1 : \tau, \cdots, f_n : \tau_n\}$ である. また, 型システムの定義より $\mathcal{L} \vdash \Gamma, c\cdot\Delta \rhd B_1 : \tau, c \in dom(\Theta), r \in dom(h)$ である. ここで, $\mathcal{L} \vdash h_1 : H_1$ とすると, $\mathcal{L}, H_1 \models r : c$ である. また H_1 は H の拡張である. **補題 1** より成り立つ.

$B = \texttt{getfield}(c, f)\cdot B_1$ **の場合.**

型システムの定義より, ある c_0, Δ' が存在して $\Delta = c_0\cdot\Delta'$ かつ $c_0 <: c$ である. またこのとき, ある r_0, S' が存在して $S = r_0\cdot S', \mathcal{L}, H \models r_0 : c_0, \mathcal{L}, H \models S' : \Delta'$ である. 遷移規則より, $(r_0\cdot S', E, M\{\texttt{getfield}(c, f)\cdot B_1\}, D_1), h) \rightarrow (v_1\cdot S', E, M\{B_1\}, D_1), h)$ かつ $v_1 = h(r_0).f$ である. 型システムの定義より, ある τ_1 が存在して, $\mathcal{L} \vdash \Gamma, \tau_1\cdot\Delta' \rhd B_1 : \tau$ かつ $\tau_1 = \Theta.c.\texttt{fields}.f$ である. ここで, $c_0 <: c$ より $h(r).f = \Theta.c.\texttt{fields}.f$ である. よって, $\mathcal{L}, H \models v_1 : \tau_1$ であるから成り立つ.

$B = \texttt{putfield}(c, f)\cdot B_1$ **の場合.**

型システムの定義より, ある τ_0, c_0, Δ' が存在して, $\Delta = \tau_0\cdot c_0\cdot\Delta', c_0 <: c, \tau_0 <: \Theta.c.\texttt{fields}.f$ である. またこのとき, ある v_0, r_0, S' が存在して $S = v_0\cdot r_0\cdot S', \mathcal{L}, H \models v_0 : \tau_0, \mathcal{L}, H \models r_0 : c_0$ である. 遷移規則より, $(v_0\cdot r_0\cdot S', E, M\{\texttt{putfield}(c, f)\cdot B_1\}, D_1), h) \rightarrow (S', E, M\{B_1\}, D_1), h\{r_0 \mapsto Update(h, r_0, f, v_0)\}$ である. 型システムの定義より, $\mathcal{L} \vdash \Gamma, \Delta' \rhd B_1 : \tau$. よって成り立つ.

$B = \texttt{invoke}(c, m)\cdot B_1$ **の場合.**

型システムの定義より, ある $\tau_1, \cdots, \tau_n, r, \Delta'$ が存在して $\Delta = \tau_n\cdot\cdots\cdot\tau_1\cdot r\cdot\Delta'$ である. またこのとき, ある $v_1\cdots, v_n, r, S'$ が存在して $S = v_n\cdot\cdots\cdot v_1\cdot r\cdot S'$ かつ $\mathcal{L}, H \models \tau_n\cdot\cdots\cdot\tau_1\cdot r\cdot\Delta' : v_1\cdots, v_n, r, S'$ である. 遷移規則より $(v_n\cdot\cdots\cdot v_1\cdot r\cdot S', E, M\{\texttt{invoke}(c, m)\cdot B_1\}, D), h \rightarrow (\phi, E_1, M_1\{M_1.entry\}, (S', E, M\{B_1\})\cdot D), h$, $E_1 = TopEnv(max(c, m))\{0 \mapsto r, 1 \mapsto v_1, \cdots, n \mapsto v_n\}, \Theta.c.\texttt{methods}.m = \{\tau_1, \cdots, \tau_n\} \Rightarrow \tau_1$. $\tau_1 =$ void のとき, 型システムの定義より, $\mathcal{L} \vdash \Gamma, \Delta' \rhd B_1 : \tau$ であるから, ダンプの型付け定義より, $H \models (S', E, M\{B_1\})\cdot D :$ void である. また, $\vdash \Pi : \Theta$ より, ある $\mathcal{L}_1$ が存在して $\vdash M_1 : \mathcal{L}_1, \mathcal{L}_1 \vdash Top(max_{entry})\{0 \mapsto c_1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}, \phi \rhd M_1.entry :$ void . このとき, $\mathcal{L}_1, H_1 \models TopEnv(max(c, m))\{0 \mapsto r, 1 \mapsto v_1, \cdots, n \mapsto v_n\} : Top(max_{entry})\{0 \mapsto c_1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}$ である. よって成り立つ. $\tau_1 \neq$ void のとき, 型システムの定義より, $\mathcal{L} \vdash \Gamma, \tau_1\cdot\Delta' \rhd B_1 : \tau$ であるから, ダンプの型付け定義より, $H \models (S', E, M\{B_1\})\cdot D : \tau_1$ である. また, $\vdash \Pi : \Theta$ より, ある $\mathcal{L}_1$ が存在して $\vdash M_1 : \mathcal{L}_1, \mathcal{L}_1 \vdash Top(max_{entry})\{0 \mapsto c_1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}, \phi \rhd M_1.entry : \tau_1$. このとき, $\mathcal{L}_1, H_1 \models TopEnv(max(c, m))\{0 \mapsto r, 1 \mapsto v_1, \cdots, n \mapsto v_n\} : Top(max_{entry})\{0 \mapsto c_1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}$ である. よって成り立つ.

第5章　型推論によるバイトコードベリファイア

本章では型推論アルゴリズムを構築する。

バイトコードベリファイアは型システムにおける型チェックに相当する。前章で定義した型システムによってプログラムの型チェックを行なう場合、次の2点を考慮する必要がある。

- **ブロックは一つの型しか持てない**
  前章で定義した型システムは単相型システムであり, 多相型を扱う機構が存在しない. これはサブルーチンの任意の場所から呼び出すことができるという多相的な性質に反する.
- **ブロックの型付け規則は他のブロックの型を参照する**
  このため, ブロックの型チェックを行なうためにはあらかじめそれが参照するラベルの型を推論しておく必要がある.

サブルーチンの多相型を表現するため、JVM のメソッドを ML における多相的 let 式によく似た項とみなして拡張する。この拡張の下では JVM のバイトコードベリファイアのアルゴリズムは型推論アルゴリズムとして記述することが可能になる.

5.1　多相型システム

定義した型システムでは, サブルーチンは $\langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle$ の形をした唯一の型を持つ. つまり, サブルーチンの呼び出し時のローカル変数およびスタックの状態は必ずそれぞれ Γ, Δ を満たさなければならない. これは, [9] がサブルーチンは内部で使用しないローカル変数については多相的であると指摘したように, サブルーチンの持つ多相性に反する. この制限を取り除き, サブルーチンに多相性をもたせるため, 型変数 t, ローカル環境変数 γ, スタック変数 δ を導入して, 型, ローカル環境, スタック環境を次のように拡張する

$$\tau ::= t \mid \mathsf{int} \mid \mathsf{void} \mid c \mid \alpha_l \mid \top \quad \Gamma ::= \phi \mid \gamma \mid \tau{\cdot}\Gamma \quad \Delta ::= \phi \mid \delta \mid \tau{\cdot}\Delta$$

この拡張の下では, 例えば, 与えられたスタックの先頭の要素を取り除くサブルーチン, およびスタックの先頭の整数の和を求めるサブルーチンの型は次のように与えられる.

$$(\alpha_l = \gamma, \delta \rhd t \text{ in } \langle \alpha_l // \gamma, t{\cdot}\delta \rhd t \rangle)$$
$$(\alpha_l = \gamma, \mathsf{int}{\cdot}\delta \rhd t \text{ in } \langle \alpha_l // \gamma, \mathsf{int}{\cdot}\mathsf{int}{\cdot}\delta \rhd t \rangle)$$

これによりサブルーチンを多相型として表現することができるが, JVM では継承の概念があるために型変数が持つことのできる型は暗黙的に制約を受けることになる. この制約を表現するために型変数の境界環境 $\mathcal{K}$ を導入する. $\mathcal{K}$ は型変数 t からその境界への関数であり, 型変数がその境界のサブクラス以外の型は持てないことを意味する. 境界はクラス名あるいは $*$ のどちらかであり, $*$ はその型変数がなんら制約を受けないことを表す. この定義の下 τ が $\mathcal{K}$ の下 c のサブクラスであることを $\mathcal{K} \vdash \tau <: c$ と書く.

この多相型をもつサブルーチンを型システムに取り込むためにサブルーチンをブロックによって使用される相互に再帰的な関数とみなし, JVM-のプログラム $\{l_b^1 : B_1, \cdots, l_b^n : B_n \mid l_s^1 : SB_1, \cdots, l_s^m : SB_m\}$ を次のように考える.

$$\begin{array}{c} \texttt{letrec}\ \ l_s^1 = SB_1 \ldots\ \ \texttt{and}\ \ l_s^m = SB_m\ \ \texttt{in} \\ \texttt{in rec}\ \ l_b^1 = B_1 \ldots\ \ \texttt{and}\ l_b^n = B_n\ \ \texttt{end} \end{array}$$

ここで letrec は多相的 let 束縛を表し, rec は単相的な再帰束縛を表す. この考えの下, メソッド M は $M^b = \{l_b^1 = B_1, \cdots, l_b^n = B_n\}$ および $M^s = \{l_s^1 : SB_1, \cdots, l_s^m : SB_m\}$ からなるラムダ計算における let 式に似た項

$$\texttt{let}\ \ M^s\ \texttt{in}\ \ M^b$$

とみなすことが出来る. この改良に伴い, ラベル環境 $\mathcal{L}$ をサブルーチンラベル環境 $\mathcal{L}_s$ およびブロックラベル環境 $\mathcal{L}_b$ に分割する. 多相型を持つサブルーチンをブロックから使用するためには, サブルーチンの型をその使用に応じて具体的な型に変換する必要がある. A が多相型を持つ B の型変数に型を代入して得られる単相型であるとき, A は B の例と呼び $A \leq B$ と書く. またサブルーチン型を σ とするとき, σ 内の全ての型変数を $\forall$ 記号によって限量化することを $Cls(\mathcal{K}, \sigma)$ と書く. このとき, 型変数は $\mathcal{K}$ が示す境界付きで限量化されることに注意する. 例えば $\mathcal{K} = \{t \mapsto c\}$ ならば,

$$Cls(\mathcal{K}, \langle \alpha_l // t{\cdot}\gamma, \alpha_l{\cdot}\delta \rhd \mathsf{int} \rangle) = \forall t <: c.\forall\gamma.\forall\delta.\langle \alpha_l // t{\cdot}\gamma, \alpha_l{\cdot}\delta \rhd \mathsf{int} \rangle)$$

以上の JVM-の構文上の拡張により, 前章で定義したブロックの型付け規則は全て境界環境 $\mathcal{K}$ の下で定義される. 例えば $\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau$ なる型判定は $\mathcal{K}, \mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau$ となる. 変更される規則はサブクラス関係を制約に持つ invoke, putfield, getfield および, サブルーチン呼び出しを行なう jsr のみである. これらの改良された型付け規則を図 5.1 に示す. 同様の変更はサブルーチンの型付け規則に対しても行なわれる.

$$\frac{\mathcal{L} \vdash \Gamma, \Theta.c.\texttt{fields}.f{\cdot}\Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, c_0{\cdot}\Delta \rhd \texttt{getfield}(c, f){\cdot}B : \tau}\ (\text{if } \mathcal{K} \vdash c_0 <: c)$$

$$\frac{\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau}{\mathcal{L} \vdash \Gamma, \tau_0{\cdot}c_0{\cdot}\Delta \rhd \texttt{putfield}(c, f){\cdot}B : \tau}\ (\text{if } \mathcal{K} \vdash c_0 <: c \wedge \mathcal{K} \vdash \tau_0 <: \Theta.c.\texttt{fields}.f)$$

$$\frac{\mathcal{K}, \mathcal{L} \vdash \Gamma, \tau_0{\cdot}\Delta \rhd B : \tau\ \ \Theta.c_1.\texttt{methods}.m = \{\tau_1', \cdots, \tau_n'\} \Longrightarrow \tau_0, \tau_0 \neq \mathsf{void}}{\mathcal{K}, \mathcal{L} \vdash \Gamma, \tau_n{\cdot}\cdots{\cdot}\tau_1{\cdot}\tau_0{\cdot}\Delta \rhd \texttt{invoke}(c_0, m){\cdot}B : \tau}$$

$$(\text{if } \mathcal{K} \vdash \tau_0 <: c_0 \wedge \mathcal{K} \vdash \tau_i <: \tau_i' \text{ for all } 1 \leq i \leq n)$$

$$\frac{\mathcal{K}, \mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau\ \ \Theta.c_1.\texttt{methods}.m = \{\tau_1', \cdots, \tau_n'\} \Longrightarrow \mathsf{void}}{\mathcal{K}, \mathcal{L} \vdash \Gamma, \tau_n{\cdot}\cdots{\cdot}\tau_1{\cdot}\tau_0{\cdot}\Delta \rhd \texttt{invoke}(c_0, m){\cdot}B : \tau}$$

$$(\text{if } \mathcal{K} \vdash \tau_0 <: c_0 \wedge \mathcal{K} \vdash \tau_i <: \tau_i' \text{ for all } 1 \leq i \leq n)$$

$$\mathcal{L} \vdash \Gamma_1, \Delta_1 \rhd \texttt{jsr}(l_1, l_2) : \tau$$

$$(\text{if } \mathcal{L}(l_1) \leq (\alpha_{l_1} = \Gamma_1, \Delta_2 \rhd \tau \text{ in } \langle \alpha_{l_1} // \Gamma_1, \alpha_{l_1}{\cdot}\Delta_1 \rhd \tau \rangle) \wedge \mathcal{L}(l_2) = \Gamma_1, \Delta_2 \rhd \tau)$$

図 5.1: 拡張された型付け規則

以上の定義を元に, メソッド M に対する型付け規則は次のように改良される.

$$\frac{\mathcal{K} \vdash M^s : \mathcal{L}^s \quad Cls(\mathcal{K}, \mathcal{L}^s) \vdash M^b : \mathcal{L}^b}{\mathcal{K} \vdash \texttt{let}\ \ M^s \texttt{in}\ \ M^b : \mathcal{L}^b}$$

ここで, $Cls(\mathcal{K}, \mathcal{L}^s)$ は $\{l : Cls(\mathcal{K}, \mathcal{L}^s(l)) \mid l \in dom(\mathcal{L}^s)\}$ を表し, $\mathcal{K}, \mathcal{L}^s \vdash M^b : \mathcal{L}^b$ は以下のように定義される.

$$\mathcal{K}, \mathcal{L}^s \vdash M^b : \mathcal{L}^b \iff \begin{array}{l} dom(M^b) = dom(\mathcal{L}^b) \text{ and} \\ \text{for any } l \in dom(M^b),\ \mathcal{L}^s \cup \mathcal{L}^b \vdash \Gamma, \Delta_l, M\{M(l)\} \rhd \tau_l \end{array}$$

5.2 型推論アルゴリズム

前節で拡張したメソッドの型付け規則はラムダ式における多相的 let 式に対する型付け規則と同じ形をしいるため, 同様の手法によってメソッドの型推論アルゴリズムを構築することができる. すなわち,

1. サブルーチンラベル環境を推論して記録する.
2. ブロックラベル環境を推論する. ブロックからサブルーチンラベルを参照するときは, 先に得られたサブルーチンラベル環境から該当するサブルーチンをインスタンス化して使用する.

5.2.1 単一化アルゴリズム

一般に, 型推論アルゴリズムは単一化アルゴリズムを基礎とする. JVM の場合, 型変数は境界条件を持つためにその単一化アルゴリズムは型変数環境 $\mathcal{K}$ の下で定義される.`Unify` を単一化アルゴリズムとすると, `Unify` は型の組からなる集合 E と型変数環境 $\mathcal{K}$ を受け取り, E の単一化である代入 S を返す. このとき, 任意の $t \in dom(\mathcal{K})$ について $\mathcal{K} \vdash S(t) <: \mathcal{K}(t)$ を満たさなければならない. 以上の考えより, `Unify` は [3] を拡張して下の変形規則により定義される.

1. $(E \cup \{(\tau, \tau)\}, S, \mathcal{K}) \Longrightarrow (E, S, \mathcal{K})$
2. $(E \cup \{(t, c)\}, S, \mathcal{K}) \Longrightarrow ([c/t]E, \{(t, c) \cup [c/t]S, \mathcal{K}) \quad (\text{if } c <: \mathcal{K}(t) \text{ or } \mathcal{K}(t) = *)$
3. $(E \cup \{(t_1, t_2)\}, S, \mathcal{K}) \Longrightarrow \left\{ \begin{array}{ll} ([t_2/t_1]E, \{(t_1, t_2) \cup [t_2/t_1]S, \mathcal{K}) & (\text{if } \mathcal{K}(t_1) = * \text{ or } \mathcal{K}(t_2) <: \mathcal{K}(t_1)) \\ ([t_1/t_2]E, \{(t_2, t_1) \cup [t_1/t_2]S, \mathcal{K}) & (\text{if } \mathcal{K}(t_2) = * \text{ or } \mathcal{K}(t_1) <: \mathcal{K}(t_2)) \end{array} \right.$
4. $(E \cup \{(t, \tau)\}, S, \mathcal{K}) \Longrightarrow ([\tau/t]E, \{(t, \tau) \cup [\tau/t]S, \mathcal{K}) \quad (\text{if } \mathcal{K}(t) = *)$

この変形規則から `Unify` は以下の関数として定義できる.

$$\mathsf{Unify}(E, \mathcal{K}) = \left\{ \begin{array}{ll} S & ((E, \phi, \mathcal{K}) \Longrightarrow^* (\phi, S, \mathcal{K}) \text{ のとき}) \\ failure & (\text{それ以外}) \end{array} \right.$$

ここで, 関係 $\Longrightarrow^*$ は関係 $\Longrightarrow$ の反射的推移的閉包である. さらに, `JVM-`ではローカル変数およびスタックに関しても型変数が存在するため, `Unify` はこれらに対しても拡張されるものとする. 例えば, $\mathcal{K}(t) = *$ ならば, $\mathsf{int}{\cdot}c{\cdot}\phi$ と $t{\cdot}\delta$ の二つのスタックを単一化すると, $[\mathsf{int}/t]$ および $[c{\cdot}\phi/\delta]$ なる代入が得られる.

5.2.2 メソッドの型推論アルゴリズム $\mathcal{WJ}$

図 5.2 にメソッドの型推論アルゴリズム $\mathcal{WJ}$ を示す. $\mathcal{WJ}$ は次のようなアルゴリズムである.

1. 型変数から構成されるサブルーチンラベルの骨格 $\mathcal{L}_s$ を生成する.
2. サブルーチンラベル環境の型推論アルゴリズム $\mathcal{WS}$ を使って, $\mathcal{L}_s$ の代入関数を得る.
3. ブロックラベル環境の骨格 $\mathcal{L}$ を作成する. このとき, エントリーブロックは Θ で示されるメソッドの型で初期化しておく.

ht

$\mathcal{WJ}(\{l_1^s = SB_1(entry_1), \ldots, l_k^s = SB_k(entry_k); l_1^b = B_1, \ldots, l_n^b = B_n\})$
  let $\mathcal{K} = \phi$
    $\mathcal{S}_i = (\alpha_{entry_i} = t_{i_1}^1 \cdot \ldots \cdot t_{i_{max}}^1 \cdot \phi; \delta_i^1 \rhd t$ in $\langle \alpha_{entry_i} // t_{i_1}^2 \cdot \ldots \cdot t_{i_{max}}^2 \cdot \phi; \delta_i^2 \rhd t \rangle)$
      $(1 \leq i \leq k)$
    $\mathcal{K} = \mathcal{K}\{t_{i_1}^1 \mapsto *, \cdots, t_{i_{max}}^1 \mapsto *, t_{i_1}^2 \mapsto *, \cdots, t_{i_{max}}^2 \mapsto *, t \mapsto *\}$ $(1 \leq i \leq k)$
    $\mathcal{L} = \{l_1^s = \mathcal{S}_1, \ldots, l_k^s = \mathcal{S}_k\}$
    $S_0 =$ the empty substitution
    $S_i = (\mathcal{WS}(S_{i-1}(\mathcal{L}), S_{i-1}(\mathcal{S}_i), SB_i)) \circ S_{i-1}$ $(1 \leq i \leq k)$ (* サブルーチンラベル環境の推論 *)
    $\mathcal{B}_{entry} = \tau_1 \cdot \ldots \cdot \tau_n \cdot \top_{n+1} \cdot \ldots \cdot \top_{i_{max}} \cdot \phi; \phi \rhd t_0$
      (where $\Theta.c.\texttt{methods}.l_{entry} = \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau_0$)
    $\mathcal{B}_i = t_{i_1}^1 \cdot \ldots \cdot t_{i_{max}}^1 \cdot \phi; \delta_i^1 \rhd t_i$ $(1 \leq i \leq n)$
    $\mathcal{K} = \mathcal{K}\{t_{i_1} \mapsto *, \cdots, t_{i_{max}} \mapsto *, t \mapsto *, t_0 \mapsto \tau_0\}$ $(1 \leq i \leq n)$
    $\mathcal{L}' = \{l_1^s = Cls(\mathcal{K}, S_k(\mathcal{S}_1)), \ldots, l_k^s = Cls(\mathcal{K}, S_k(\mathcal{S}_k));$
      $l_{entry} = \mathcal{B}_{entry}, l_1^b = \mathcal{B}_1, \ldots, l_n^b = \mathcal{B}_n\}$
    $S_0' =$ the empty substitution
    $S_i' = (\mathcal{WB}(S_{i-1}'(\mathcal{L}'), S_{i-1}'(\mathcal{B}_i), B_i)) \circ S_{i-1}'$ $(1 \leq i \leq k)$ (* ブロックラベル環境の推論 *)
  in $S_n'(\mathcal{L}')$

図 5.2: 型推論アルゴリズム $\mathcal{WJ}$

4. 推論された $\mathcal{L}_s$ の下, ブロックラベル環境の型推論アルゴリズム $\mathcal{WB}$ によってラベル環境 $\mathcal{L}_b$ の代入関数 S を得る.

5. S に $\mathcal{L}_b$ を適用することで推論されたラベル環境を得る.

ここで i_{max} はメソッドで使用されるローカル変数の数を表す. また型変数環境 $\mathcal{K}$ はグローバルな環境とし, このアルゴリズムを通して連続的にアクセスができるものとする.

5.2.3 ラベル環境の型推論アルゴリズム $\mathcal{WS}, \mathcal{WB}$

一般にプログラムの型推論は, プログラムを構成する個々のプログラムの型を推論し, その型の間の制約を方程式として記述してそれを単一化アルゴリズムによって求めることによって行なわれる. 同様の考えの下, ブロックの型推論は図 5.3 に示すように,

1. ブロックを構成するコード列 A を先頭の命令 I と残りのコード列 B に分割する.
2. B の型推論を行ない, 推論された型と I の間の制約条件を記述しそれを単一化アルゴリズムで求める

がその基本的なアルゴリズムである. これは再帰関数によって容易に実現できる. 図 5.4, 図 5.5 にサブルーチンブロックおよびベーシックブロックの推論アルゴリズム $\mathcal{WS}$, $\mathcal{WB}$ の一部を示す. $\mathcal{WS}$ および $\mathcal{WB}$ はラベル環境, ブロックの型判定, ブロックのコード列を受け取り, 型変数の代入関数を返す.

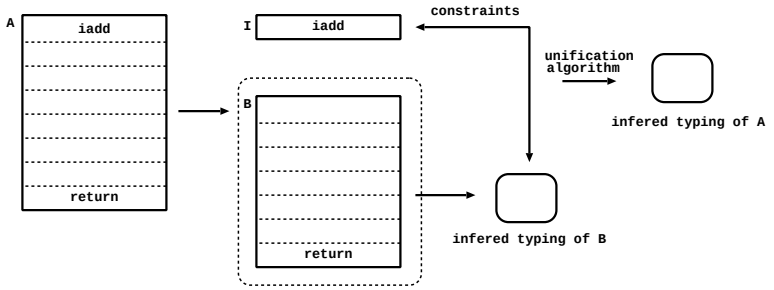

図 5.3: ブロックの型推論

$$\mathcal{WS}(\mathcal{L}, (\alpha_l = \Gamma_1; \Delta_1 \rhd \tau \text{ in } \langle \alpha_l // \Gamma_2; \Delta_2 \rhd \tau \rangle), \texttt{ret}(i))$$
$$= \mathsf{Unify}\{(\Gamma_2(i), \alpha_l), (\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\}$$
$$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma_1; \Delta_1 \rhd \tau \rangle \Gamma_2; \Delta_2 \rhd \tau, \texttt{aload}(i) \cdot SB) =$$
$$\text{let } \mathcal{K} = \mathcal{K}\{t \mapsto \texttt{Object}\} \ (* \textit{ any class is a subclass of } \texttt{Object} \textit{ class } *)$$
$$S_1 = \mathsf{Unify}\{(\Gamma_2(i), t)\}$$
$$S_2 = \mathcal{WS}(S_1(\mathcal{L}), S_1(\langle \alpha_l // \Gamma_1; \Delta_1 \rhd \tau \rangle \Gamma_2; t \cdot \Delta_2 \rhd \tau), SB)$$
$$\text{in } \ S_2 \circ S_1$$
$$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma_1; \Delta_1 \rhd \tau \rangle \Gamma_2; \Delta_2 \rhd \tau, \texttt{goto } l' \cdot SB) = \text{let } (\langle \alpha_{l'} // \alpha_{l'} \rangle \Gamma_2'; \Delta_2' \rhd \tau) = \mathcal{L}(l')$$
$$\text{in if } l = l' \text{ then } \mathsf{Unify}\{(\Gamma_2, \Gamma_2'), (\Delta_2, \Delta_2')\}$$
$$\text{else } \textit{failure}$$

図 5.4: サブルーチンラベル環境の型推論アルゴリズム

$$\mathcal{WB}(\mathcal{L}, \Gamma; \Delta \rhd \tau, \texttt{ireturn}) = \mathsf{Unify}\{(\Delta, \mathsf{int} \cdot \delta), (\tau, \mathsf{int})\}$$

$$\mathcal{WB}(\mathcal{L}, \Gamma; \Delta \rhd \tau, \texttt{goto } l) = \text{let } \Gamma'; \Delta' \rhd \tau' = L(l)$$
$$\text{in } \ \mathsf{Unify}\{(\Gamma, \Gamma'), (\Delta, \Delta'), (\tau, \tau')\}$$

$$\mathcal{WB}(\mathcal{L}, \Gamma; \Delta \rhd \tau, \texttt{jsr}(l_1, \ l_2)) =$$
$$\text{let } (\alpha_{l_1} = \Gamma_1; \Delta_1 \rhd \tau_1 \text{ in } \langle \alpha_{l_1} // \Gamma_2; \Delta_2 \rhd \tau_2 \rangle) = \mathit{FreshInst}(L(l_1))$$
$$\Gamma'; \Delta' \rhd \tau' = L(l_2)$$
$$\text{in } \ \mathsf{Unify}\{(\Gamma, \Gamma_2), (\alpha_{l_1} \cdot \Delta, \Delta_2), (\Gamma', \Gamma_1), (\Delta', \Delta_1), (\tau, \tau_2), (\tau', \tau_1)\}$$

$$\mathcal{WB}(\mathcal{L}, \Gamma, \Delta \rhd \tau, \texttt{getfield}(c, \ f) \cdot B) =$$
$$\text{let } \mathcal{K} = \mathcal{K}\{t \mapsto c\}$$
$$\tau' = \Theta.c.\texttt{fields}.f$$
$$S_1 = \mathsf{Unify}\{(\Delta, t \cdot \delta)\}$$
$$S_2 = \mathcal{WB}(S_1(\mathcal{L}), S_1(\Gamma, \tau' \cdot \delta \rhd \tau), B)$$
$$\text{in } \ S_2 \circ S_1 \ \mathcal{WB}(\mathcal{L}, \Gamma; \Delta \rhd \tau, \texttt{invoke}(c, \ m) \cdot B) =$$

図 5.5: ブロックラベル環境の型推論アルゴリズム

第6章　実装と評価

前章で示した型推論アルゴリズムを実装し, バイトコードベリファイアを作成した. システムは ML で書かれており, いくつかのモジュールの集合からなる. 本章では, 実際のシステムのソースコードを示しながら, その構造の詳細を説明する.

さらに, いくつかのクラスファイルを使用して本システムのベリファイアの能力を検証する.

6.1　構造

実装したバイトコードベリファイアの構造を図 6.1 に示す.

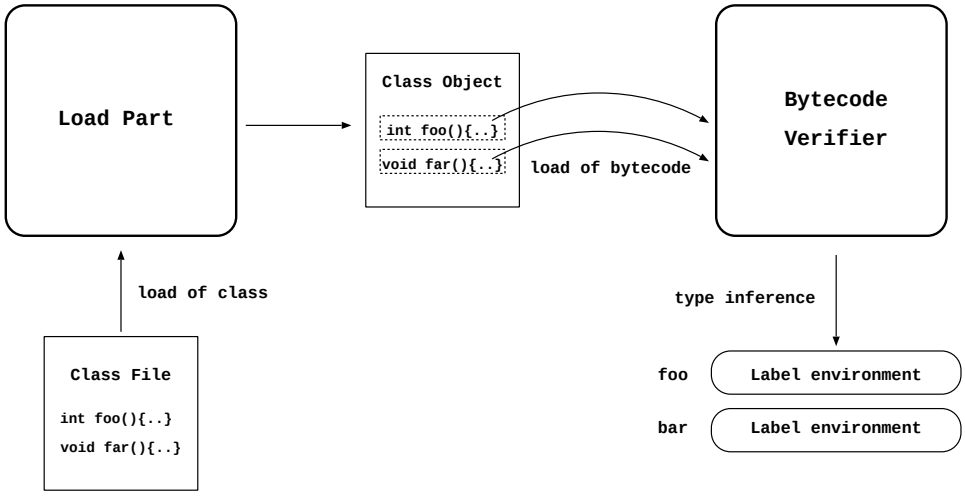

図 6.1: 型推論システム

システムは大きく分けて, クラスファイルのロード部分と推論部分から構成される. ロード部はクラスファイルを読み込みその内部表現であるクラスオブジェクトを生成する. 推論部分はクラスオブジェクトから全てのメソッドを読み込み, そのラベル環境を推論する. クラスファイルの形式は仕様で厳密に定義されており, いくつかの構造体が連続的に配置された構造をとる. ロード部はそれらの構造体をクラスファイルの先頭から順番に読み込み内部形式に変換する. その実装は単純であり, 技術的に目を引く点はほとんどない. よって, 以降では推論部に焦点をしぼってその構造の詳細を説明する.

推論部は前章で示された型推論アルゴリズムを実現する部分であり, 主に `Types`, `TypesUtil`, `Env`, `CodeUtil`, `Kind`, `Unify`, `Infer`, `Top` の 8 つのモジュールから構成される. それらの相互関係は図 6.2 のようになる.

`Top` はトップレベル関数を提供する. トップレベルでは, クラスオブジェクトをロードして全てのメソッドのラベル環境を推論して表示する. もし推論に失敗すれば, そのメソッド名を表示して終了する. 個々のメソッドのラベル環境の推論は `Infer` が担当する.`Infer` は `CodeUtil` が提供する関数によってまずメソッド

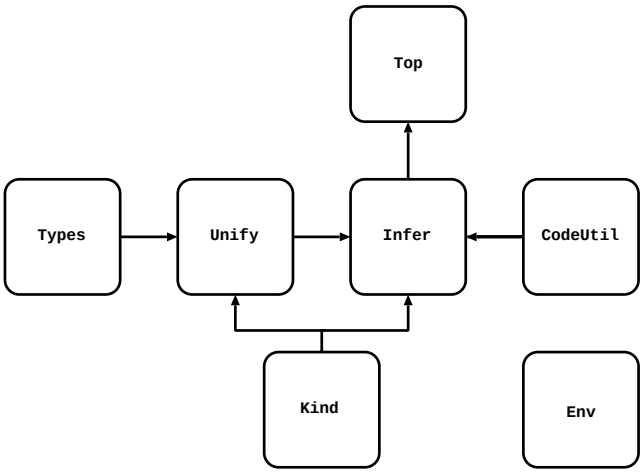

図 6.2: 推論部分のモジュール構成

のコード配列をコードブロックの集合に分割する. その後, 前章のアルゴリズムにしたがってラベル環境を推論する.Unify は Infer で使用される単一化関数を提供する.Kind は型変数環境を実現するモジュールであり, Unify および Infer によって使用される.Types, TypesUtil は種々のデータ型およびその操作関数を提供する.Env はデータを保存するための環境およびその操作関数を提供する汎用のモジュールであり, 全てのモジュールから使用することができる.

以下, それぞれのモジュールの詳細についてソースコードの一部を参考にしながら説明する.

6.1.1 Types モジュール

JVM の型, ローカル変数, スタックを表現するデータ型を定義する.

```
(* jvmtypes *)
type var = string
type label = string
datatype jvmtypes =
    JTvar of var                    (* type variable *)
  | JTint                           (* integer type *)
  | JTvoid                          (* void type *)
  | JTclass of string               (* object type *)
  | JTret of label                  (* return address type *)
  | JTnotuse                        (* unuseable local variable type *)
(* type sequence *)
datatype typeseq =
    TSempty                         (* empty sequence *)
  | TSvar of var                    (* sequence variabel *)
  | TScons of jvmtypes * typeseq    (* cons constructor *)
type envtype = typeseq              (* local environment type *)
type stktype = typeseq              (* stack type *)
(* block and subroutine typing *)
datatype typing =
    Block of envtype * stktype * jvmtypes
  | Sub of jvmtypes * (envtype * stktype * jvmtypes) * (envtype * stktype * jvmtypes)
```

型 jvmtypes は JVM の型である. スタックおよびローカル変数は前章で定義したようにどちらも同じ構

造をしており, 型の列を表す typeseq によって表現する. 型 typing はコードブロックおよびサブルーチンブロックの型付け判定を表すデータ型である.

6.1.2 TypesUtil モジュール

Types で定義した型に関する操作関数を提供する. 主な関数は下に示す型変数の置換を行なう関数である.

```
(* subst v t T => [t/v]T *)
fun subst v t T =
    case T of
        JTvar v' => if v = v' then t else T
      | _ => T
and substSeq v t T =
    case T of
        TScons(t', s) => TScons(subst v t t', substSeq v t s)
      | _ => T
fun substSeqVar sv st T =
    case T of
        TSempty => T
      | TSvar _ => if sv = T then st else T
      | TScons(t, s) => TScons(t, substSeqVar sv st s)
```

subst は型変数を型で置換する.substSeq は型列中の型変数を与えられた型によって置換する. さらに substSeqVar は, 列型変数を列型で置換する関数である.

6.1.3 Env モジュール

種々のインデクス付けされたデータを保存する環境とそれを操作するいくつかの関数を提供する. 環境を表すデータ型は次のように表される. 内部的には効率が悪いが次のようにリストで表現している.

```
datatype ('key, 'value) env = Env of ('key * 'value) list
```

'key, 'value はそれぞれインデクスとデータを表す. 環境の操作関数の一部を示す.

```
fun lookup(Env ([]), _) = raise NotThere
  | lookup((Env ((name, value)::es)), name') =
    if name = name' then value else lookup(Env (es), name')
fun update(Env ([]), name, value) = Env ([(name, value)])
  | update((Env ((name, value)::es)), name', value') =
    if name = name' then Env ((name, value')::es)
    else Env ((name, value)::(explode (update(Env (es), name', value'))))
fun mapVal (Env e) f = Env (map (fn (x, y) => (x, f y)) e)
fun foldEnv f z (Env ((x, y)::xs)) = f ((x, y), foldEnv f z (Env xs))
  | foldEnv f z (Env []) = z
```

foldEnv はリストの fold 関数同様の処理を行なう.

6.1.4 CodeUtil モジュール

メソッドのバイトコード配列をブロックの集合に変換する関数 codeToBlocks を提供する. Java バイトコードでは, ラベルは相対アドレスで表現されているため, codeToBlocks は最初にコード列をスキャンしてこれを絶対アドレスに変換する. 次に, 得られたコード列をコードブロックの集合に分割する. 次の 4 つのパスによってコード列をコードブロックの集合に変換する.

1. ベーシックブロックのラベルを収集する

2. サブルーチンブロックのラベルを収集する

3. ラベル集合を元にコードブロックの集合を求める

4. サブルーチンラベル集合を元にサブルーチンブロックの集合を求める

結果的に複数回コード列をスキャンすることになり効率が悪いが, アルゴリズムが簡潔でプログラム化しやすいためこの方式で行なう.
ベーシックブロックのラベル集合は次の関数によって得られる.

```
fun getLabels (c::cs) labels =  (l, codes)::blocks
    case c of
        Bgoto l => if isExist(l, labels) then labels
                   else let val code' = getCodeFromLabel code l
                        in getLabels code' (l::labels)
                        end
      | Breturn => labels
      | Bireturn => labels
      | Bareturn => labels
      | Bifeq l => if isExist(l, labels) then getLabels cs labels
                   else let val code' = getCodeFromLabel code l
                            val labels' = getLabels code' (l::labels)
                        in getLabels cs labels'
                        end
      | Bifne l => if isExist(l, labels) then getLabels cs labels
                   else let val code' = getCodeFromLabel code l
                            val labels' = getLabels code' (l::labels)
                        in getLabels cs labels'
                        end
      | Bjsr(_, l) => getLabels cs (l::labels)
      | _ => getLabels cs labels
```

得られたラベル集合を元にコードブロックを生成する. それぞれのラベルについて, ラベル位置からコードをスキャンしながらコードブロックを生成していく. コードブロックの生成はコードブロックの構造にしたがって下の関数によって行なわれる.

```
fun getBlock (c::cs) block =
    case c of
        Breturn => (block@[c])
      | Bireturn => (block@[c])
      | Bareturn => (block@[c])
      | Bgoto _ => (block@[c])
      | Bjsr _ => (block@[c])
      | _ => getBlock cs (block@[c])
```

全てのラベルにこの関数を適用することで, コードブロックの集合を得る.
サブルーチンラベルの収集はそれぞれが属するエントリラベルを区別する必要があるため, 次の 2 段階で行なわれる. 最初にサブルーチンのエントリラベルを以下の関数によって収集する.

```
fun getEntryLabels [] entryLabels = entryLabels
  | getEntryLabels (c::cs) entryLabels =
    case c of
       Bjsr(l, _) => if isExist(l, entryLabels) then getEntryLabels cs entryLabels
                     else getEntryLabels cs (l::entryLabels)
      | _ => getEntryLabels cs entryLabels
```

次に, それぞれのエントリラベルについてそれに属するサブルーチンラベルを収集する. 個々のラベル集合を集める関数を下に示す.

```
fun getSubLabels (c::cs) labels =
    case c of
        Bgoto l => if isExist(l, labels) then labels
                   else let val code'' = getCodeFromLabel code' l
                        in getSubLabels code'' (l::labels)
                        end
      | Breturn => labels
      | Bireturn => labels
      | Bareturn => labels
      | Bifeq l => if isExist(l, labels) then getSubLabels cs labels
                   else let val code'' = getCodeFromLabel code' l
                            val labels' = getSubLabels code'' (l::labels)
                        in getSubLabels cs labels'
                        end
      | Bifne l => if isExist(l, labels) then getSubLabels cs labels
                   else let val code'' = getCodeFromLabel code' l
                        val labels' = getSubLabels code'' (l::labels)
                        in getSubLabels cs labels'
                        end
      | Bret _ => labels
      | Bjsr(_, l) => if isExist(l, labels) then labels
                      else getSubLabels cs (l::labels)
      | _ => getSubLabels cs labels
```

全てのエントリラベルについて上記の関数を適用してサブルーチンのラベル集合が得られる. サブルーチンラベルの集合は $\{(e,(l,\cdots)),\cdots,(e,(l,\cdots))\}$ のような構造となる. ここで, e はエントリラベル, l はサブルーチンラベルである. このラベル集合を元にサブルーチンブロックの集合を求める. 基本的な考え方はコードブロックの場合と同じであるためソースコードは省略する.

6.1.5 Kind モジュール

Kind モジュールは前章で述べた型変数環境 $\mathcal{K}$ とその操作関数を提供する.$\mathcal{K}$ は次のように定義される.

```
datatype kind =
    Bottom
  | Bound of string
type kindEnv = (var, kind) env
val kindenv = ref (emptyEnv()) : kindEnv ref
```

kind は型変数の境界を表すデータ型である. kindenv は $\mathcal{K}$ に相当し, 変数と kind の組からなる環境で表される. 前章のアルゴリズムで示したように, kindenv はグローバルな変数として用いられ, Unify, Infer それぞれのモジュールから参照される.

6.1.6 Unify モジュール

単一化関数 unify を提供する. 前章で述べたように, 単一化は型のみならず, ローカル環境およびスタックについても行なわれる. そのため, 型を単一化する関数 unifyType および型列を単一化する関数 unifySeq を内部関数として持つ.
unifyType のコードを下に示す.

```
fun unifyType([], S) = S
  | unifyType(((T as JTvar v, t)::E), S) =
    let val bound = lookupKind(!kindenv, v)
        val E' = substPairs (subst T t) E
        val S' = substPairs (subst T t) S
    in
       case t of
          JTvar v' =>
             let val bound' = lookupKind(!kindenv, v')
             in
               case bound of
                   Bottom => if v = v' then unifyType(E, S)
                             else (kindenv := removeKind(!kindenv, v);
                                   unifyType(E', ((T, t)::S')))
                 | Bound name =>
                      (case bound' of
                           Bound name' =>
                               if isSubClass(name, name') then
                                  let val env = removeKind(!kindenv, v)
                                  in (kindenv := updateKind(env, v', bound);
                                      unifyType(E', ((T, t)::S')))
                                  end
                               else if isSubClass(name', name) then
                                       (kindenv := removeKind(!kindenv, v);
                                        unifyType(E', ((T, t)::S')))
                                    else raise UnifyFail
                         | Bottom => let val env = removeKind(!kindenv, v)
                                     in (kindenv := updateKind(env, v', bound);
                                         unifyType(E', ((T, t)::S')))
                                     end)
             end
        | JTclass name =>
             (case bound of
                  Bottom => (kindenv := removeKind(!kindenv, v);
                             unifyType(E', ((T, t)::S')))
                | Bound name' => if isSubClass(name, name') then
                                    (kindenv := removeKind(!kindenv, v);
                                     unifyType(E', ((T, t)::S')))
                                 else raise UnifyFail)
        | _ => (kindenv := removeKind(!kindenv, v);unifyType(E', (T, t)::S'))
   end
```

これは関数のコードの一部であり, プログラムの読み易さのため一部の例外処理は省略している. unifyType は型の組のリスト E と空のリストを受け取り, E を単一化する代入を表す S を返す. 代入 S は型の組のリストで表現される. 型変数の単一化を行なう場合はその境界条件をチェックし, 満たした場合のみ成功する. 単一化が成功して不要となった型変数は型変数環境から除去する.

型の変数には jvmtypes 型の型変数と typeseq 型の型変数があるため, 型列の単一化を行なうためにはそれぞれの型変数を置換する代入を求めなければならない. 例えば, int·c·Δ および t·δ の二つのスタックを単一化する場合, 型変数 t は c に置換され, スタック変数 δ は c·Δ に置換される. 下に型列の単一化をおこなう unifySeq を示す.

```
fun unifySeq([], TS, SS) = (TS, SS)
  | unifySeq(((T as TSvar _, t)::E), TS, SS) =
    let val E' = substPairs (substSeqVar T t) E
        val SS' = substPairs (substSeqVar T t) SS
```

```
    in unifySeq(E', TS, ((T, t)::SS'))
    end
  | unifySeq(((t, T as TSvar _)::E), TS, SS) =
    let val E' = substPairs (substSeqVar T t) E
        val SS' = substPairs (substSeqVar T t) SS
    in unifySeq(E', TS, ((T, t)::SS'))
    end
  | unifySeq(((TScons(t1, s1), TScons(t2, s2))::E), TS, SS) =
    let val S = unifyType([(t1, t2)], [])
        val E' = substPairs (pairsToSubst S substSeq) E
        val TS' = substPairs (pairsToSubst S subst) TS
        val SS' = substPairs (pairsToSubst S substSeq) SS
   in unifySeq(((s1, s2)::E'), (S@TS'), SS')
   end
  | unifySeq(((t1, t2)::E), TS, SS) = if t1 = t2 then unifySeq(E, TS, SS)
                          else raise UnifyFail
```

ここで, parisToSubst は型の組の集合で表された代入を置換関数に変換する関数である.
unify は 3 つの引数を持つ. それぞれ型, ローカル変数, スタックの組からなるリストであり, これらをそれぞれ単一化する代入関数を返す.

```
fun unify EJ EE ES =
    let
       val SJ1 = unifyType(EJ, [])                          (* unify types *)
       val EE1 = substPairs (pairsToSubst SJ1 substSeq) EE
       val ES1 = substPairs (pairsToSubst SJ1 substSeq) ES
       val (SJ2, SE) = unifySeq(EE1, SJ1, [])                 (* unify local environment *)
       val ES2 = substPairs (pairsToSubst SJ2 substSeq) ES1
       val (SJ3, SS) = unifySeq(ES2, SJ2, [])                 (* unify stack *)
       val substSeqSet = pairsToSubst SJ3 substSeq
       val substEnvVar = pairsToSubst SE substSeqVar
       val substStackVar = pairsToSubst SS substSeqVar
    in (pairsToSubst SJ3 subst,  fn x => substStackVar (substSeqSet x),
                                 fn x => substStackVar (substSeqSet x))
    end
```

6.1.7 Infer モジュール

型推論アルゴリズム $\mathcal{WJ}$ を実装する関数 inferMethod を提供する. さらに内部関数として $\mathcal{WS}$, $\mathcal{WB}$ をそれぞれ実装した inferSub, inferBlock を持つ. inferMethod はクラス名, メソッド名, ディスクリプタ[1]を表す 3 つの文字列を引数として受け取り, そのメソッドのラベル環境を推論する関数であり, WJ アルゴリズムをほぼ忠実に実装する. 前章で示した通り, inferMethod は最初にラベル環境の骨格を生成する. 次に, サブルーチンのラベル環境を推論する. これを行なうのは下の関数である.

```
fun inferSubLabel labelEnv codeEnv S0 =
    let fun inferTyping ((l, Block _), S) = S
          | inferTyping ((l, Sub typing), S) =
            let val code = lookup(codeEnv, l)
                val labelEnv' = substLabelEnv S labelEnv
                val typing' = substTyping S (Sub typing)
                val S' = inferSub (labelEnv', typing', code)
            in composeSubst(S', S) end
    in Env.foldEnv inferTyping S0 labelEnv end
```

ここで, labelEnv, codeEnv はそれぞれラベル環境とコードブロック環境を表し, ラベルをキーとする env 型によって与えられる.

[1]ディスクリプタはクラスファイルで用いられる, フィールドおよびメソッドの型を表す文字列である

次に, 推論されたサブルーチンラベル環境を元に, ベーシックブロックのラベル環境を推論する. これは以下の関数によって行なわれる.

```
fun inferBlockLabel labelEnv codeEnv S0 =
    let fun inferTyping ((_, Sub _), S) = S
          | inferTyping ((l, Block typing), S) =
            let val code = lookup(codeEnv, l)
                val labelEnv' = substLabelEnv S labelEnv
                val typing' = substTyping S (Block typing)
                val S' = inferBlock (labelEnv', typing', code)
            in composeSubst(S', S) end
    in Env.foldEnv inferTyping S0 labelEnv end
```

サブルーチンブロックの型推論を行なう inferSub を下に示す. ここに示すのは aload の場合のみである.

```
fun inferSub (labelEnv,  Sub (ret, typing, (e, s, r)),  Baload n::SB) =
    let val typvar = jvmVar(Bound "Object")
        val S1 = unify [(typvar, lookupLocalEnv(e, n))] [] []
        val subTyping = Sub (ret, typing, (e, TScons(typvar, s), r))
        val S2 = inferSub (substLabelEnv S1 labelEnv,  substTyping S1 subTyping,  SB)
    in composeSubst(S2, S1) end
```

jvmVar は境界付きのフレッシュな型変数を生成する関数であり, 同時に型変数環境に変数と境界を登録する.

ベーシックブロックの型判定を推論する関数 inferBlock を以下に示す.getfield の場合のみを示す.

```
fun inferBlock (labelEnv,  Block (e, s, r),  Bgetfield(class, field, desc)::B) =
    let val fieldType = descToType desc
        val typvar = jvmVar(Bound class)
        val stkvar = stackVar()
        val S1 = unify [] [] [(s, TScons(typvar, stkvar))]
        val typing = Block (e, TScons(fieldType, stkvar), r)
        val S2 = inferBlock (substLabelEnv S1 labelEnv,  substTyping S1 typing,  B)
    in composeSubst(S2, S1) end
```

6.1.8 Top モジュール

Top は, メイン関数である verifier を提供する. verifier はクラスファイル名を受け取り, クラスファイル中の全てのメソッドをサーチして, それぞれのラベル環境を Infer で提供される inferMethod により求める.

6.2 実装環境および使用したツール

システムの構築および評価は Linux 上で行なった. 実装言語は関数型言語の一種である Standard Ml of New Jersey である. クラスファイル生成および Sun のベリファイアを検証するために, Sun の Java コンパイラおよび JVM を使用した. また, 評価を行なう際, クラスファイルを改変するために D-java および jasmin を用いた.D-java は複数の出力表現をサポートしたクラスファイルの逆アセンブラである. jasmin は JVM のアセンブラ言語の一種であり, jasmin 形式のプログラムからクラスファイルを生成することができる. 表.6.1 にこれらのツール類の詳細をまとめる.

ツール	目的
Linux カーネルバージョン 2.2.16-1k7	実装環境
Standard ML of New Jersey バージョン 110.0.6	実装言語
Sun Java コンパイラ バージョン 1.3.0	クラスファイル生成
Sun Java Virtual Machine バージョン 1.3.0	ベリファイアの検証
D-java および jasmin バージョン 1.05	クラスファイル改変

表 6.1: 実装環境およびツール類

6.3 評価

この節では実際のクラスファイルを使用してシステムの能力を検証する. クラスファイルの生成には Sun の Java コンパイラを使用し, 意図するクラスファイルを表す Java ソースプログラムを記述するのが困難な場合は djava および jasmin を使用して手書きでクラスファイルを生成した.

6.3.1 実行例１

下の Java のソースプログラムを Java コンパイラによってプログラムをコンパイルすると, 3 つのクラスファイル `A.class`, `B.class`, `sample1.class` が生成される.

```
class sample1 {
    int foo(A a, B b) {
        int sum;
        try { } finally {
                sum = a.bar(3) + b.bar(2);
                if (sum == 0) return 1;}
        return sum; }
}
class A { int bar(int n) { return n; } }
class B extends A { }
```

*sample*1 は `try-finally` 構文を持つメソッド `foo` を有する. Java コンパイラは `try-finally` 構文の `finally` 節をサブルーチンにコンパイルする.
システムは最初にメソッド `foo` のバイトコード配列を次のようなコードブロックの集合に変換する.

```
Entry:  jsr(S1, L1)

L1:     goto(L2)

L2:     iload(3)
        ireturn

S1:     astore(5)
        aload(1)
        iconst(3)
        invokevirtual(A, bar, (I)I)
        aload(2)
        iconst(2)
        invokevirtual(A, bar, (I)I)
        iadd
        istore(3)
```

```
          iload(3)
          ifne(S2)
          iconst(1)
          ireturn

   S2(S1): ret(5)
```

実際のコード列には例外ハンドラが含まれるが, ここでは無視するものとする. S1, S2(S1) はサブルーチンブロックを表し, S2(S1) は S2 が S1 に属するサブルーチンであることを意味する. システムは, それぞれのコードブロックのラベル環境を推論して次のような結果を表示する.

```
   Entry  : {CL(sample1), CL(A), CL(B), -, -, -}{} => INT
   L1     : {CL(sample1), CL(A), CL(B), INT, -, ret(S1)}{} => INT
   L2     : {CL(sample1), CL(A), CL(B), INT, -, ret(S1)}{} => INT

   S1     :  (ret(S1), {('a<*), ('b<A), ('c<A), INT, ('d<*), ret(S1)}{'L} => INT
              // {('a<*), ('b<A), ('c<A), ('e<*), ('d<*), ('f<*)}{ret(S1), 'L} => INT)
   S2(S1) :  (ret(S1), {('a<*), ('b<A), ('c<A), INT, ('d<*), ret(S1)}{'L} => INT
              // {('a<*), ('b<A), ('c<A), INT, ('d<*), ret(S1)}{'L} => INT)
```

ここで, 'a は型変数を表し, ('a<A) は'a が境界 A のサブクラスであることを意味する. サブルーチン内部にメソッド呼び出しが存在するため, サブルーチンの型に制約付きの型変数が存在している。

6.3.2 実行例２

次のコードは正しいプログラムであるが, Sun のベリファイアではエラーとなる.

```
Entry : iconst_n(1)
        jsr(S1, L1)

L1 :    istore_(1)
        aload_(0)
        jsr(S1, L2)

L2 :    astore_(0)
        goto(L3)

L3 :    return

S1 :    astore_(2)
        ret(2)
```

サブルーチン S1 は, Entry ブロックおよび L1 ブロック双方から呼び出されている. 最初の呼び出しの場合, スタックのトップには整数がプッシュされ、次の呼び出し時にはオブジェクト参照がプッシュされている。Sun の仕様では、サブルーチンは内部で使用しなローカル変数については多相性を実現しているが、スタックについては考慮されていないためエラーとなる。
我々のシステムでは次のようなラベル環境を推論する.

```
   Entry : {CL(sample4), -, -}{} => VOID
   L1    : {CL(sample4), -, ret(13)}{INT} => VOID
   L2    : {CL(sample4), INT, ret(13)}{CL(sample4)} => VOID
   L3    : {CL(sample4), INT, ret(13)}{} => VOID

   S1    :   (ret(13), {('q<*), ('r<*), ret(13)}{'F} => ('m<*)
               // {('q<*), ('r<*), ('s<*)}{ret(13), 'F} => ('m<*))
```

6.3.3 実行例３

以下に示すのは, 整数を二つ引数にとり整数型を返すメソッドのコードである.

```
Entry : iconst_n(1)
        istore_(3)
        iload_(1)
        ifeq(L1)
        aload_(0)
        astore_(3)
        goto(L1)

L1 :    iload_(1)
        iload_(2)
        iadd
        goto(L2)

L2 :    ireturn
```

このコードは Sun のベリファイアでは検証可能であるにも関わらず, 我々のシステムではエラーとなる. ブロック L1 へのジャンプは 2 つの場所から生じている. 最初の場所では, ローカル変数 3 には整数が代入されており, 次の場所では, ローカル変数 3 にはオブジェクト参照が代入されている. したがって, ブロックが単相的な我々のシステムでは型付けすることができない. Sun のベリファイアアルゴリズムでは排反する型は使用不可能な型とみなし, このローカル変数にアクセスがあったときのみエラーとするため、このようなコードも検証可能となる。

第7章　結論

本章では本論文の成果を述べ, 今後望まれる研究について言及して本論文のまとめとする.

7.1　研究の成果

本論文では, JVM に対する型システムを構築して JVM の操作的意味論に基づいて健全性の証明を行なった。さらにサブルーチンの多相型を表現するために型システムを拡張し、プログラムを ML の多相的 let 式に類似した項と解釈することにより、型推論アルゴリズムを構築することができた。。さらに型推論アルゴリズムを実装し、本研究が示す型理論的なアプローチが JAVA のベリファイアを系統的に実装する基礎として有効であることを実証した。

7.2　今後の課題

本研究で定義した型システムは, 従来のプログラミング言語における型システムと同様の能力を持つシステムであり, そこで研究されてきた種々のテクニックを JVM に取り入れることが期待される. 例えば,

- **高階のメソッドを持つ JVM**
  型付きラムダ計算では型に関数型を含めることで, 関数を引数にとる関数や関数を返す関数である高階の関数を提供している. 高階関数は複雑なプログラムを簡潔に書くことが出来るため, 同様の機構が JVM に対しても有用であると思われる. 高階のメソッドは型に $\Gamma, \phi \triangleright \tau$ の形をしたメソッド型を導入して型システムを拡張すると容易に実現可能である.

- **より柔軟な型推論アルゴリズム**
  今回定義した型システムは, サブルーチンを多相型として表現することには成功したが, ブロックは単相型であり, また, サブルーチン内部からのサブルーチン呼び出しは, 単相的に型付けされる. 従って, 前章でみたように, Java のベリファイアでは検証可能なコードが我々のシステム我々のシステムでは検証出来ないということが起こってしまう. そのため, より強い多相性を型システムに取り入れることが望まれる.

- **型推論アルゴリズムの健全性および完全性の証明**
  型推論アルゴリズムが健全であるとは, 型推論アルゴリズムが推論した型判定はすべて型システムが導出可能であるという性質である. 型推論アルゴリズムが完全であるとは, アルゴリズムがプログラムの持つ全ての型判定を推論可能であることを示す性質である. これらは型推論アルゴリズムが満たすべき性質である。本論文ではこれらについては証明していないが、型システムの健全性の証明同様に従来の型理論的な手法により容易に証明できると思われる。

本論文では JVM のサブセットに対して型システムを定義したため、いくつかの JVM の特徴を考慮していない。しかし、今回除外したこれらの特徴は、以下に述べるような手法によって解決可能であると思われる。

- **メソッドのオーバーロード**
  メソッドのオーバーロードとは, 一つのクラス内に型が異なる同じ名前のメソッドを定義可能な機構であり、メソッドはそのシグネチャ(名前と型) によって識別される。本研究で定義した JVM-ではメソッド呼び出し命令である `invoke`(c, m) はクラス名とメソッド名のみを引数にとるため、メソッドのオーバーロード機構は含まれないが、引数にメソッド型を含めることで容易に実現可能である。

- **例外**
  例外は Java の try-catch 構文を実現する機能であり, JVM では例外ハンドラがその処理を行なう. 例外ハンドラは catch 節に相当するコード列であり、try 節内の任意の場所で例外が発生すると例外ハンドらに制御が移行する。つまり、例外ハンドラも一つのコードブロックであり、多相性を有するため、多相型として型システムに取り入れることが可能であると思われる。

- **インターフェース**インターフェースはメソッド定義を持たない型情報のみを持つクラスである. 本論文では Θ に相当する部分である. インターフェースの導入により, クラスの型付けを行なうためにはメソッドの型付けに加えてそのスーパークラスであるインターフェースとの整合性をチェックする必要がある。

謝辞

本研究を進めるにあたり、毎週御指導して頂きました大堀淳教授に感謝致します。また、日々の研究を様々な面からサポートし有意義なものとしてくれた研究室の皆様に感謝致します。尚、本研究の型推論の部分は大堀淳教授との共同研究の成果によるものであり、この場を借りてお礼申し上げます。

関連図書

[1] S. Freund and J. Mitchell. A type system for object initialization in the Java byte code language. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 310–328, 1998.

[2] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–166, 1999.

[3] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2):203–260, 1989.

[4] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine Subroutines. In *Proceedings of Static Analysis Symposium*, pages 17–32. Springer-Verlag, Berlin Germany, 1998.

[5] S. Katsumata and A. Ohori. Proof-directed de-compilation of low-level code. In *European Symposium on Programming, Springer LNCS 2028*, pages 352–366, 2001.

[6] X. Leroy. *Polymorphic typing of an algorithmic language*. PhD thesis, University of Paris VII, 1992.

[7] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 70–78, 1999.

[8] A Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proceedings of International Symposium on Functional and Logic Programming*, 1999.

[9] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 149–160, 1998.

[10] T.Lindholm and F.Yellin. *The Java virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999.