9 情技応第 888 号

高度情報化支援ソフトウェア育成事業

自己反映計算に基づく Java 言語用の

開放型 Just-in-Time コンパイラ OpenJIT の研究開発

機能仕様書

平成 11 年 1 月

富士通株式会社

目次

図目次

第 1 章

技術開発概要

1.1 技術開発の背景（課題設定）

デジタル技術が普遍性を持つ今日，従来の計算技術は急速に陳腐化し，新たな計算環境に適した汎用性のある技術を我が国が研究開発することが大いに求められている．例えば，広域ネットワーク，マルチメディア環境，NC，並びに電子商取引などの新しいアプリケーションにおいては，可搬性の高いプログラムが要求されており，各国ともその技術開発に凌ぎを削っている．特に，インターネット及びイントラネットを中心とした互換性が要求される環境，あるいは組み込み機器のように計算資源が限定されている環境においては，Java 言語に代表される機器間で高い可搬性を有する言語が重要視されてきている．

Java 言語では，バイトコードのコンパクトでかつ可搬性のあるプログラムの中間形式から，必要な部分を実行時にネイティブコードにコンパイルし，実行速度を向上させる Just-In-Time (JIT) コンパイラが開発されているが，技術フレームワークの欠如，JIT 自身の可搬性の欠如，最適化技術の未発達を含む問題が指摘されている．たとえば，「最適化」は通常速度の最適化であり，限られたメモリやその他の計算資源のもとで，最大の効率を得るという，組込み型のアプリケーションで必要な “Resource-Efficient”(資源効率の高い) 計算の最適化はなされない．さらに，今後様々な計算環境へ適合するため，(1) 個々のアプリケーション及び計算環境に特化した最適化と，(2) 計算環境とアプリケーションに応じたコンパイルコードの拡張が必要になってくるが，従来型の JIT は (1) は汎用性のある最適化しか行わず，また，(2) に対しては，言語の拡張や新規の機能に対応してコード生成の手法を変えるようなカスタム化は不可能でり，Java の利便性と性能に関して大いに妨げとなっている．

1.2 前提とする理論・技術

OpenJIT は，富士通と大学側双方が保有するコンパイラ処理系理論を活用する．さらに前提として，Java における JIT コンパイラの一般的技術 (API を含む) を活用する．以下，JIT コンパイラの動作を述べ，その後本開発で用いる技術を解説する．

(1) JIT コンパイラ技術

JIT コンパイラは，古くは Lisp や Smalltalk などの，中間言語系の言語処理系を高速化させるための技術として開発された．Lisp, Smalltalk, Java では，通常の処理ではまずソースプログラムを機種間で共通な中間コード (Java の場合はバイトコードと呼ばれる) に変換する．Java の場合は，javac と呼ばれるコンパイラを用い，ソースプログラムからクラスファイルが作成される．中間コードは，通常は何らかの仮想マシンの命令列 (Java の場合は，Java Virtual Machine - JVM と呼ばれるスタックベースの仮想マシン) であり，異なるプロセッサ上ではこの仮想マシンを解釈実行するインタプリタを実装することによって，中間コードを実行する．この場合，機種間の可搬性解釈実行のオーバーヘッドが大きいため，通常のネィティブなバイナリコードを実行するコンパイラ処理系と比較すると，数倍から数十倍の速度低下が生じる．

中間言語方式の可搬性と，ネィティブコードに近い高速性を両立させるのが，Just-In-Time(JIT) コンパイラ技術である．JIT コンパイラでは，中間コードからネィティブコードへのコンパイルを実行時に必要に応じて行うことにより，インタプリタの速度を改善する．コンパイルにかかる時間と，実行時間の利得を比較して，後者の方が大きければ，全体的に速度向上が見込める．実際，プログラムの実行の大半はごく一部のコードを繰り返し実行するので，コンパイル時間の amortization(償却) によって速度が向上する．

Java の標準開発環境である JDK では，外部の JIT コンパイラを JVM に組み込める API が準備されている．基本的には，JIT は各クラスに対するメソッドディスパッチ部位を書き換え，直接メソッド本体ではなく，JIT コンパイラが起動されるようにしておく．メソッド呼び出しによって起動された JIT コンパイラは，「JIT コンパイルをして利得を得られる」と判断すると，メソッドを構成するバイトコードをその場でコンパイルし，ネィティブコードを得て，ヒープ領域に格納する．さらに，メソッドディスパッチ部位をさらに書き換え，以後の起動は直接ネィティブコードが起動されるようにする．そして，与えられた引数をネィティブコードに渡して，それを起動する．

本開発においては，上記の JIT の API を用いることにより，OpenJIT を JDK の処理系に組み込む．このように標準の API に準拠することにより，一部制限が生じるが，より広範に開発ソフトウェアを流布し，実用に供することが可能となる．

(2)　OpenJIT コンパイラに用いる富士通および大学の技術

OpenJIT が実現する Open Compiler としての性質は，自己反映計算 (リフレクション) が理論的なベースとなっている．

自己反映計算とは，計算システムが自分自身の構造と計算過程をモデル化した表現を持ち，その表現を操作することによって，自己の構造や計算過程の進行を操作することを呼ぶ．自己反映計算モデルとは，このような計算の枠組のことをいい，1982 年に Indiana 大学 (当時 MIT) の Brian Smith 教授の提唱した 3-Lisp によって最初の理論的な基盤が提唱された．基本的には，適切な計算系の定義により，通常の計算を表す従来からの ベースレベル のコードに加え，自己の計算系を表す メタレベル のコードをユーザは記述することが可能となる．

Open Compiler は，自己反映計算の理論に基づき，計算系の挙動をコンパイル時に変更する機能を提供するシステムである．コンパイラをオブジェクト指向設計に基づきモジュール化してユーザから変更可能にし，さらにコンパイル時にメタ計算をあらかじめ静的に行なうことによって，言語の拡張やアプリケーション固有の最適化などを行なう．

一方， Java の JIT コンパイラに関しては，以下のような技術的なベースがある．JVM(Java Virtual Machine) はスタックマシンであり，引数の受け渡しや途中結果を保持するためにレジスタを使用しない． JVM が使用するレジスタは４つである．このため， Intel 486 のような汎用的に使えるレジスタの数が少ないマシンでも効率よく実行できるようになっている．

しかし，一般の RISC プロセッサでは利用できる汎用レジスタは 30 個以上ある．JVM が持つレジスタをネイティブのプロセッサのレジスタに割り付けただけでは，利用しないレジスタが多くある．加えて JVM はスタックマシンにも関わらず，プログラムの正当性を確保するためにバイトコードにおいてスタックが動的に延び縮みすることを禁じている．これはベリファイアによってチェックされる．スタックの領域はメソッドが呼び出される時に必要な数だけ確保されるが，メソッド内において， VM のある命令がアクセスするスタックの位置は常に同じである．従ってスタックをレジスタに静的に割り付けることが可能である．またスタック上にとられるローカル変数もレジスタに割り付けることが可能である．そこでコンパイラがこのレジスタ割り付けを効率よく行なえば，実行速度の大幅な向上が期待できる．

また， JVM はスタックマシンのため，演算のオペランドおよびその結果は全てス

タック上に置かれる．そのため，スタック上に命令を置く命令とスタックから値を取り出す命令が通常のプロセッサにおいては冗長な命令となる．このような冗長な命令を取り除く peephole 最適化を行なうことによって更に実行速度を上げることができる．

この JIT コンパイラは C で記述され，かつ設計も Open Compiler としてのものではない．本研究開発では，この JIT コンパイラの構築経験を基盤として，新たに Open Compiler である OpenJIT のシステムを構築するものとする．

さらに，Java を実用アプリケーションとして利用するために，Java からネイティブコードを生成するコンパイラ HBC(High performance Bytecode Compiler) を作成した．このコンパイラはバイトコードを入力とし，実行可能ファイルを生成する静的コンパイラである．C，Fortran のコンパイラと同様な最適化処理および，Java 固有の最適化を実施することによって高い性能を達成できた．本研究開発では，HBC で適用された様々な標準最適化技法も OpenJIT コンパイラに適用する．

1.3 技術開発のテーマ設定

上記の問題を解決するために，我々は従来型のコンパイラ技術とは異なる，自己反映計算 (リフレクション) の理論に基づいた “Open Compiler” (開放型コンパイラ) 技術をベースとして，アプリケーションや計算環境に特化した言語の機能拡張と最適化が行える JIT コンパイラのテクノロジ “OpenJIT” を研究開発する．開発のターゲットは実用性や広範な適用性を考慮して Java 言語とするが，技術的には他の同種のプログラム言語にも適用可能である．本研究開発により，我が国がこの分野でリーダーシップをとり，我が国が得意とする組み込み機器，マルチメディア機器，並列科学技術計算などにおいて次世代の基盤技術を持つことを目標とする．

1.4 技術開発の概要

Open Compiler とは，自己反映計算 (リフレクション，Reflection) を理論的基盤として，コンパイラにさまざまな言語拡張や最適化の為のモジュールを組み込む技術基盤である．本研究では，Just-In-Time コンパイラに Open Compiler の技術を適用し，クラス単位での最適化の為のカスタマイゼーションを可能にする．これにより，アプリケーションや計算環境に特化した言語機能の拡張や，最適化を可能とすることができるようになる．

OpenJIT では，クラスライブラリの作成者がクラス単位でそのクラスに固有の最適化のモジュールを組み込むことを可能とする．これにより，様々な計算環境・プラットフォーム・アプリケーションに対する適合や，複雑な最適化を組み込むことも可能となる．

これらの最適化の一部は，従来のコンパイラでも可能ではあったが，OpenJIT でのポイントはそのような最適化をライブラリの作成者が行なえることである．これにより，従来は一般的な場合には JIT コンパイラの解析能力が低すぎて，最適化が行なえない場合も，ユーザがそのクラスに特化して指定することができるようになる．あるいは，最適化が余りにもアプリケーションに特殊すぎて，汎用の JIT コンパイラに組み込むことができない場合にも対応が可能になる．

さらに，OpenJIT 自身，Java を用い，オブジェクト指向のアプリケーションフレームワークとして構築される．これにより，以下の従来のコンパイラにはない特徴を OpenJIT コンパイラは持つこととなる．

コンパイラの実行時の変更

コンパイラ全体に対して拡張や最適化の指示を行うモジュールを実行時に動的に組み込むことが OpenJIT では可能である．これらの拡張や最適化は，Java のクラスファイルに隠蔽されるため，そのクラスライブラリやそれに基づくフレームワークを用いるユーザは，自らはさまざまなハッキングを行うことなく，自動的にその最適化の利益を享受することができる．

コンパイラ自身の動的コンパイル

OpenJIT 自身 Java で書かれているため，当初は VM でインタプリタ (解釈系) 実行されるが，OpenJIT 自身により必要な部分がコンパイルされ，高速化・最適

化がなされていく.

図 1.1に, OpenJIT コンパイラの概要図を示す.

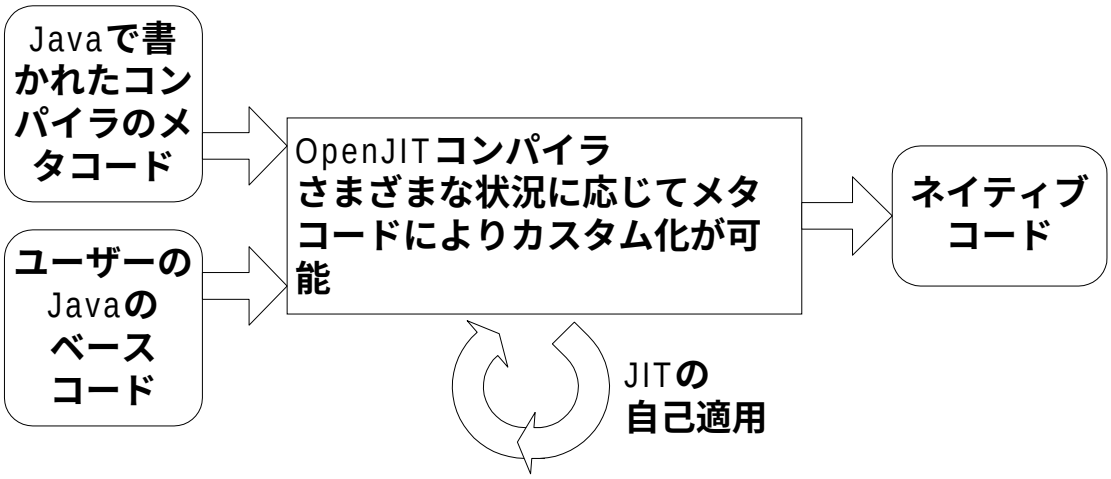

図 1.1: OpenJIT の概念図

第 2 章

全体概要

2.1 システム概要

(1) 目標及び目的

従来型のコンパイラ技術とは異なる，自己反映計算 (リフレクション) の理論に基づいた “Open Compiler”(開放型コンパイラ) 技術をベースとして，アプリケーションや計算環境に特化した言語の機能拡張と最適化が行える JIT コンパイラのテクノロジ “OpenJIT” を研究開発する．開発のターゲットは実用性や広範な適用性を考慮して Java 言語とするが，技術的には他の同種のプログラム言語にも適用可能である．

(2) 概要説明

OpenJIT の実現する JIT コンパイラとしての特徴は以下の通りである．

- JDK 1.1.4 以降の JIT インターフェースを用いて設計することにより，既存の JDK 1.1.4 以降の JavaVM と組み合わせて， JIT コンパイラとして機能するものとする．
- リフレクションの技術を用いて，クラスライブラリの作成者がクラス単位でそのクラスに固有の最適化のモジュールを組み込むことを可能とする．
- 計算環境やアプリケーションに応じたコンパイルコードの拡張も， OpenJIT の拡張機能により可能になるものとする．

- コンパイラの実行時の変更が可能になるものとする．コンパイラ全体に対して拡張や最適化の指示を行うモジュールを実行時に動的に組み込むことが可能になるものとする．
- コンパイラ自身の動的コンパイルが可能になるものとする． OpenJIT 自身が Java で書かれているため，当初は JavaVM でインタプリタ実行されるが，OpenJIT 自身により必要な部分がコンパイルされ，高速化・最適化がなされるものとする．

(3)　全体構成

OpenJIT コンパイラシステムの全体構成を，従来の JIT コンパイラとの比較と共に図 2.1に示す．

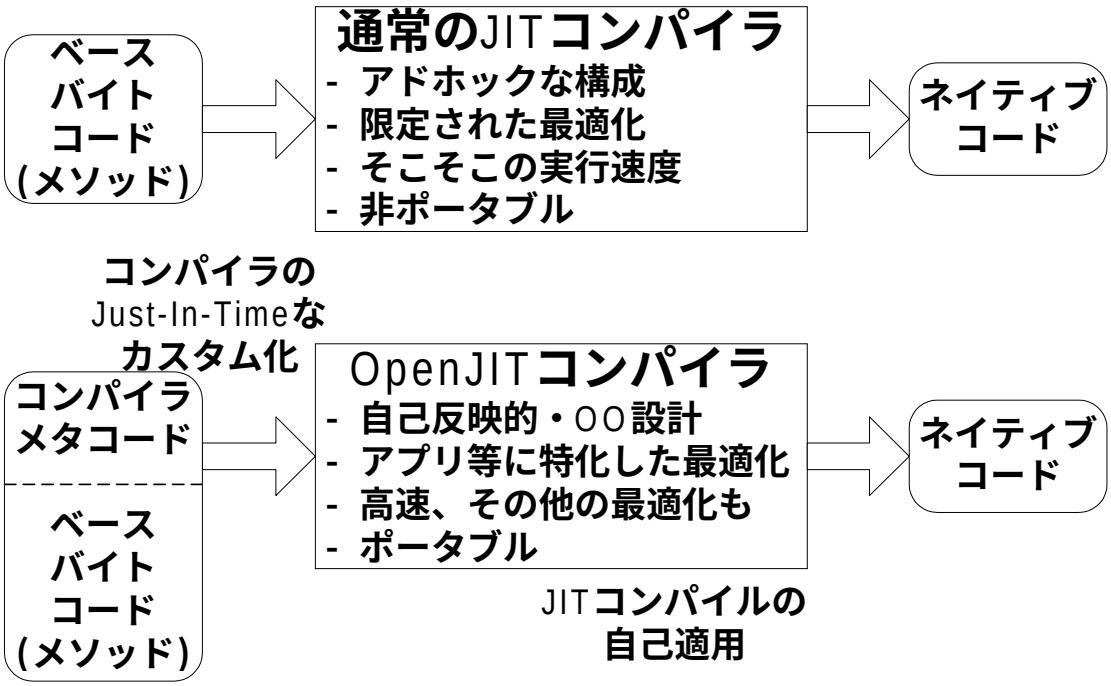

図 2.1: 従来型の JIT コンパイラと， OpenJIT コンパイラとの比較

2.2 プログラム概要

(1) 目的及び目標

システム概要で示した通り，従来型のコンパイラ技術とは異なる，自己反映計算(リフレクション)の理論に基づいた “Open Compiler”(開放型コンパイラ)技術をベースとして，アプリケーションや計算環境に特化した言語の機能拡張と最適化が行えるJIT コンパイラのテクノロジ “OpenJIT” を研究開発する．

(2) 概要説明

最適化やクラスの拡張を行うクラスライブラリの開発者は，Open Compiler の API を用いて，プログラム解析，および最適化の為のプログラム変換，さらに実行時のプローブなどの記述を，クラスのベースレベルのプログラムとは別に Java を用いて記述する．これらは，クラスライブラリの作成者により，ベースレベルのバイトコード，言語拡張部分のバイトコード，および最適化を行うためのバイトコードと，クラスファイルの言語拡張アノテーションに分類される．次に，クラスファイルのユーザがこのクラスファイルをダウンロードすると，言語拡張および最適化機能を実現するライブラリ，さらに(必要な場合は) OpenJIT の各種サポートライブラリを同時にダウンロードする．次に，OpenJIT のディスコンパイラ機能が，言語拡張モジュールと最適化機能が必要な抽象レベル(AST)までバイトコードの逆変換を行う．変換されたプログラムに対し，最適化機能がクラスに特化したプログラム解析，アノテーションの認識，および最適化変換を行う．変換後，再びそれらはバイトコードに変換され，さらに，バイトコードからネィティブコードに変換される．その際にも，JIT に peephole 最適化を行う機能を含む各種最適化機能が起動される．

(3) 開発対象範囲

OpenJIT コンパイラシステム全体の構成は図 2.2の通りである．このうち，開発対象となるものは実線の矩形の OpenJIT フロントエンドシステムと OpenJIT バックエンドシステムである．

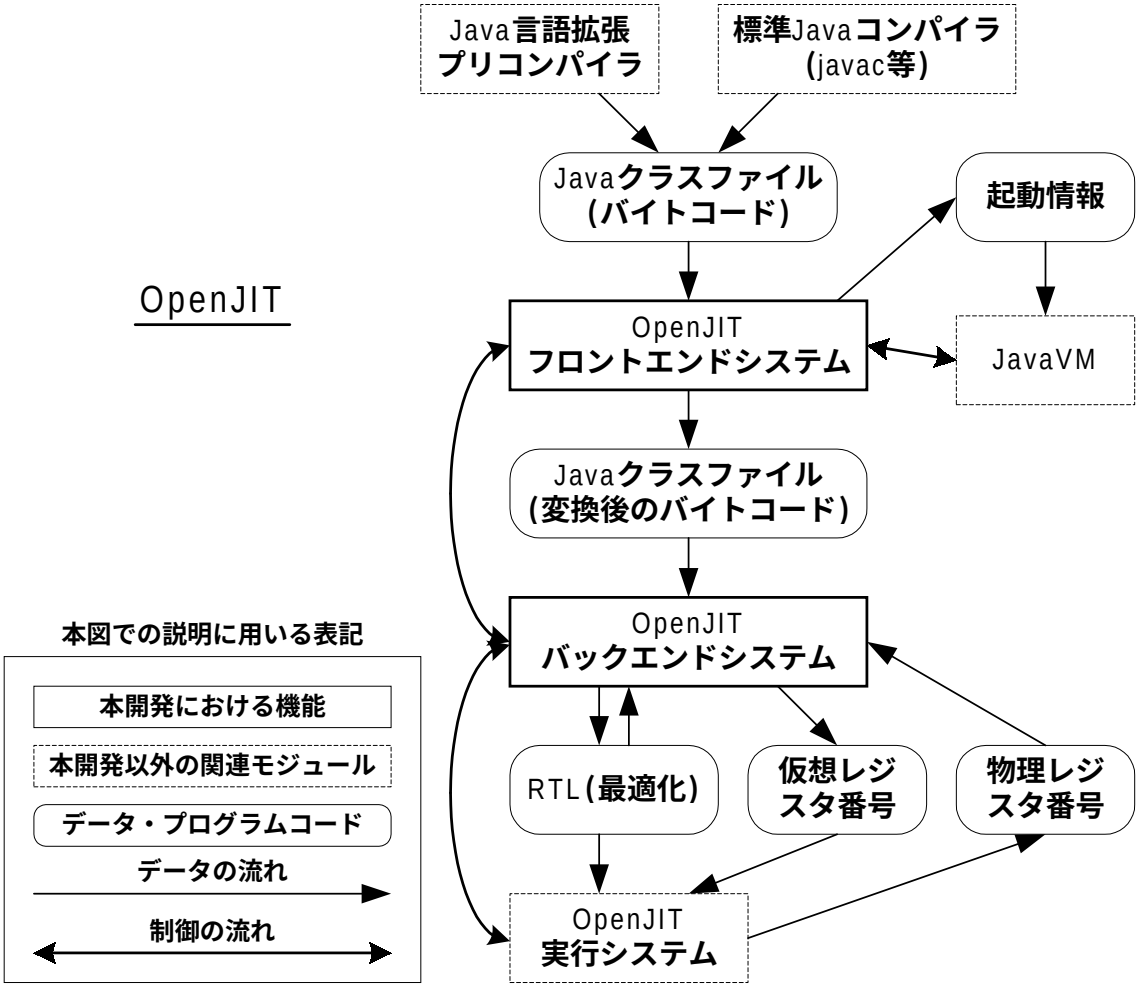

図 2.2: OpenJIT コンパイラシステムの開発対象範囲

第 3 章

設計指針

(1)　実現方式

OpenJIT コンパイラシステムは，OpenJIT フロントエンドシステムと OpenJIT バックエンドシステムから構成される．

OpenJIT フロントエンドシステムは，Java のバイトコードを入力とし，高レベルな最適化を施して，再びバイトコードを出力する．OpenJIT バックエンドシステムは，得られたバイトコードに対し，より細かいレベルでの最適化を行い，ネイティブコードを出力する．

OpenJIT コンパイラシステムが与えられたメソッドに対して起動されると，OpenJIT フロントエンドシステムは対象となるバイトコード列に対して以下の処理を行う．

まず，OpenJIT バイトコードディスコンパイラ機能は，バイトコードを逆変換して AST を出力する．この際には，与えられたバイトコード列から，元のソースプログラムから生成されるコントロールグラフのリカバリを行う．同時に，OpenJIT クラスファイルアノテーション解析機能により，このクラスファイルのアトリビュート領域に何らかのアノテーションが付記されていたときに，その情報を得る．この情報は，AST 上の付加情報として用いられる．得られた AST に対し，OpenJIT 最適化機能によって，最適化が施される．最適化に必要な情報は，OpenJIT フローグラフ構築機能，OpenJIT フローグラフ解析機能により抽出される．最適化時のプログラム変換は，OpenJIT プログラム変換機能が司って実施され，変換後のバイトコードがバックエンドシステムに出力される．

次に，OpenJIT フロントエンドシステムによって最適化されたバイトコード列に対

し，OpenJIT バックエンドシステムは，さらなる最適化処理を行い，ネイティブコードを出力する．

OpenJIT ネイティブコード変換機能はバックエンド系処理全体の抽象フレームワークであり，OpenJIT バックエンドシステムの各機能のインタフェースを定義する．このインタフェースに沿って具体的なプロセッサに応じたクラスでモジュールを記述することにより，様々なプロセッサに対応することが可能となる．

OpenJIT 中間コード変換機能によって，バイトコード列からスタックオペランドを使った中間言語へと変換を行なう．バイトコードの命令を解析して分類することにより，単純な命令列に展開を行う．得られた命令列に対し，OpenJIT RTL 変換は，このスタックオペランドを使った中間言語からレジスタを使った中間言語 (RTL) へ変換する．バイトコードの制御の流れを解析し，命令列を基本ブロックに分割する．バイトコードの各命令の実行時のスタックの深さを計算することで，スタックオペランドから無限個数あると仮定した仮想的なレジスタオペランドに変換する．次に，OpenJIT Peephole 最適化機能によって，RTL の命令列の中から冗長な命令を取り除く最適化を行ない，最適化された RTL が出力され，最後に OpenJIT SPARC プロセッサコード出力モジュールにより，SPARC プロセッサのネイティブコードが出力される．OpenJIT SPARC プロセッサモジュールは，ネイティブコード生成時のレジスタ割り付けのために OpenJIT レジスタ割付機能を利用する．出力されたネイティブコードは，JavaVM によって呼び出され実行されるが，その際に OpenJIT ランタイムモジュールを補助的に呼び出す．

OpenJIT コンパイラシステムは，Java 言語自身でポータブルに記述され，Java Virtual Machine の JIT に対する標準 API を満たすように作成される．具体的には，OpenJIT は，以下の各機能で構成される (図 3.1,3.2参照．このうち実線の長方形は今回開発される各機能を，丸まった長方形はデータを，点線は既存のモジュール，あるいは本開発をベースとして，我々が将来研究開発するモジュールを表す)．

OpenJIT フロントエンドシステム

1. OpenJIT コンパイラ基盤機能
   OpenJIT 全体の基本動作を司る．

2. OpenJIT バイトコードディスコンパイラ機能
   Java のクラスファイルに含まれるバイトコードを，いわゆる discompiler 技術に

より，コンパイラ向けの中間表現に変換する．

3. OpenJIT クラスファイルアノテーション解析機能
   プログラムグラフ (AST) に対して，コンパイル時に適切な拡張された OpenJIT のメタクラスを起動できるようにする．

4. OpenJIT 最適化機能
   各種解析モジュールおよびプログラム変換モジュールを用い，プログラム最適化を行なう．

5. OpenJIT フローグラフ構築機能
   AST およびコントロールフローグラフを受け取り，対応するデータ依存グラフ，コントロール依存グラフ，などのフローグラフを出力する．

6. OpenJIT フローグラフ解析機能
   フローグラフ構築モジュールで構築されたプログラム表現のグラフに対し，グラフ上の解析を行なう．

7. OpenJIT プログラム変換機能
   OpenJIT フローグラフ解析機能の結果やユーザのコンパイラのカスタマイゼーションに従って，プログラム変換を行なう．

OpenJIT バックエンドシステム

1. OpenJIT ネイティブコード変換機能
   OpenJIT バックエンド全体の制御を行う．

2. OpenJIT 中間コード変換機能
   バイトコードから中間言語への変換を行う．

3. OpenJIT RTL 変換機能
   中間言語から RTL への変換を行う．

4. OpenJIT Peephole 最適化機能
   RTL を最適化し，無駄な命令を取り除いて最適化を行う．

5. OpenJIT レジスタ割付機能
   仮想レジスタから物理レジスタへの割付を行う．

6. OpenJIT SPARC プロセッサコード出力モジュール
   RTL からネイティブコードへの変換を行う.

7. OpenJIT ランタイムモジュール
   ネイティブコードが実行時に呼び出し，ネイティブコードの実行ををサポートする.

注: 13, 14 は，今回開発は行われるが，契約の対象外とする.

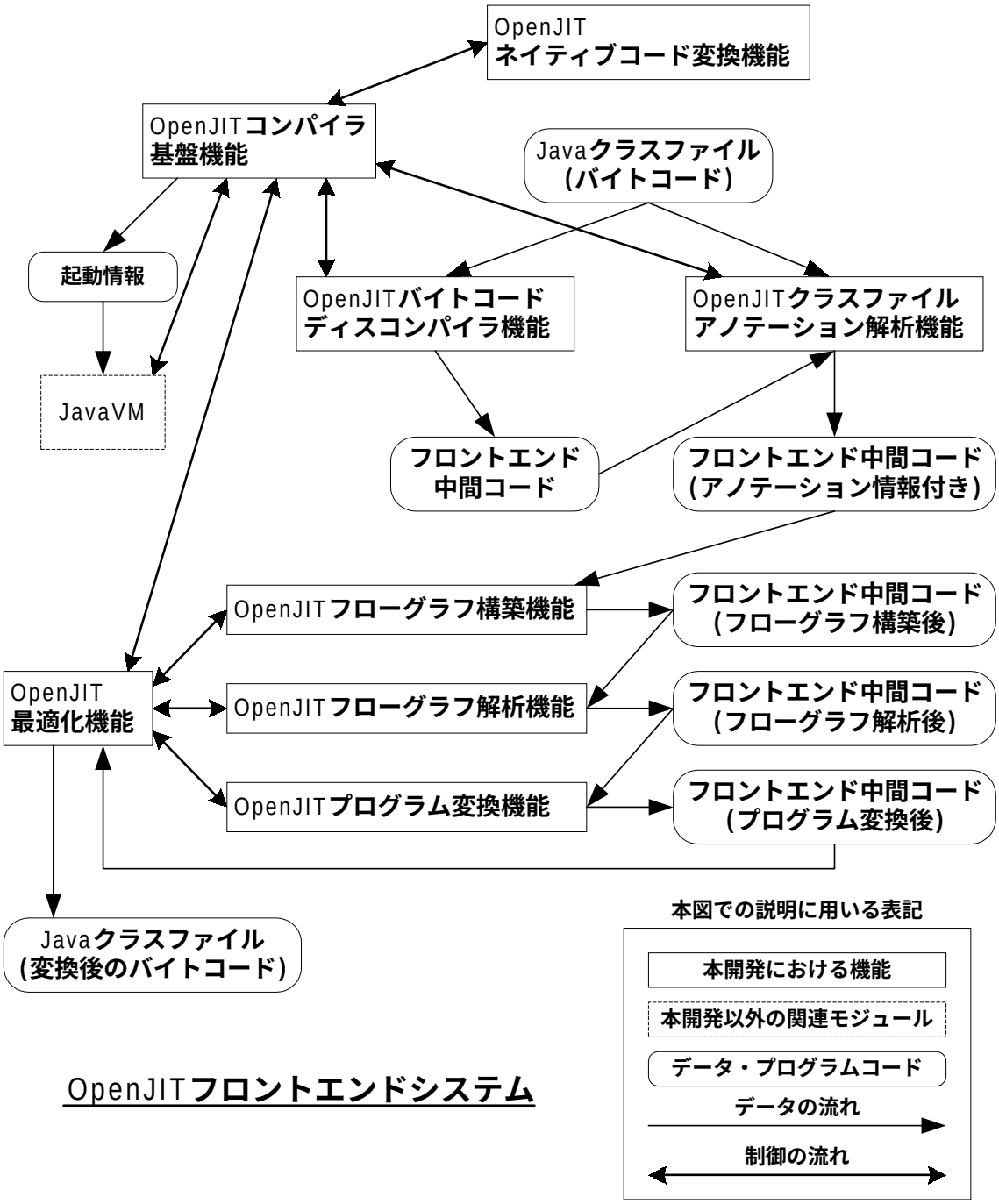

図 3.1: OpenJIT フロントエンドシステム

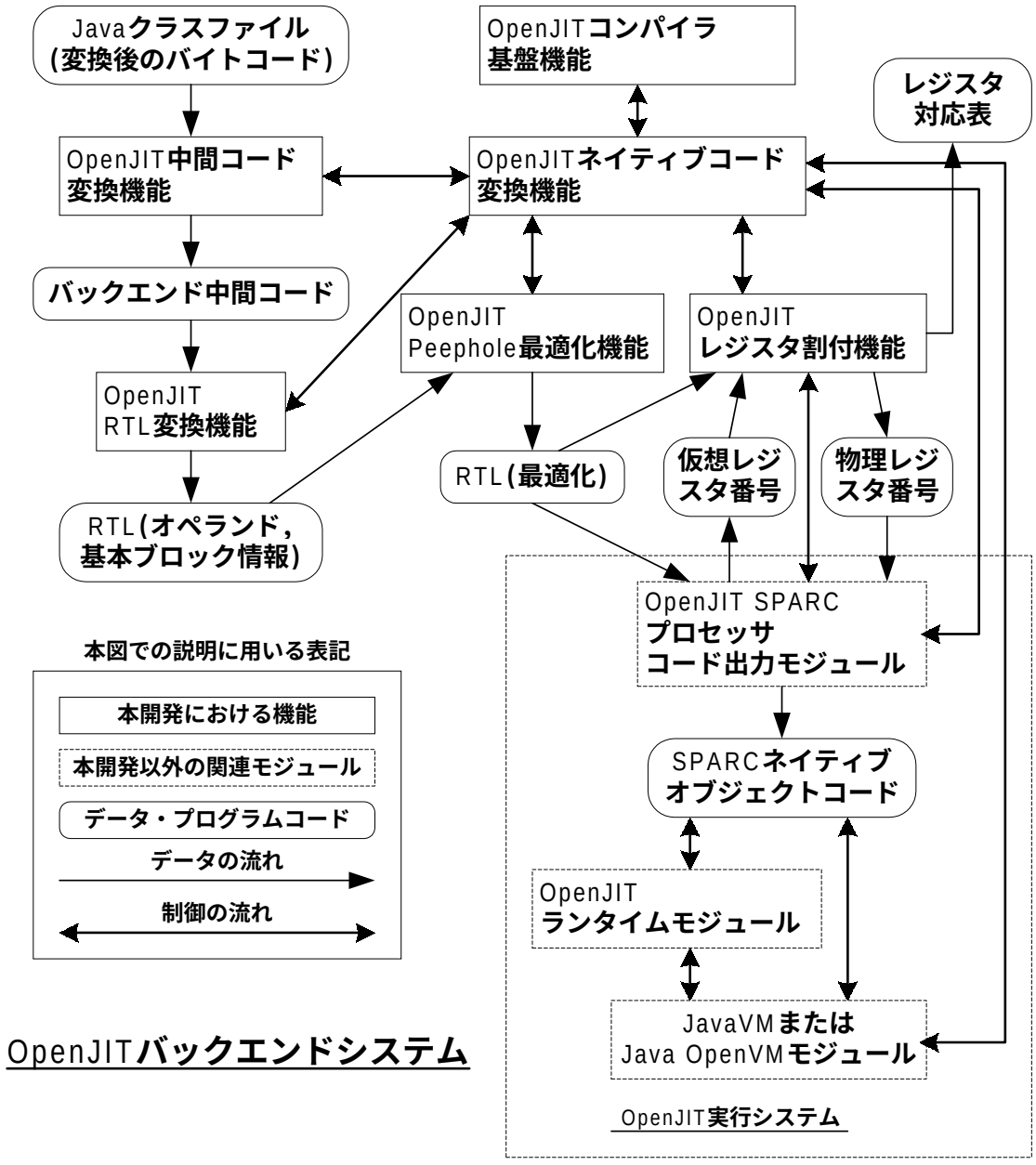

図 3.2: OpenJIT バックエンドシステム

(2)　他システムとの関連

他システムとの関連は図 3.3に示す通りである.

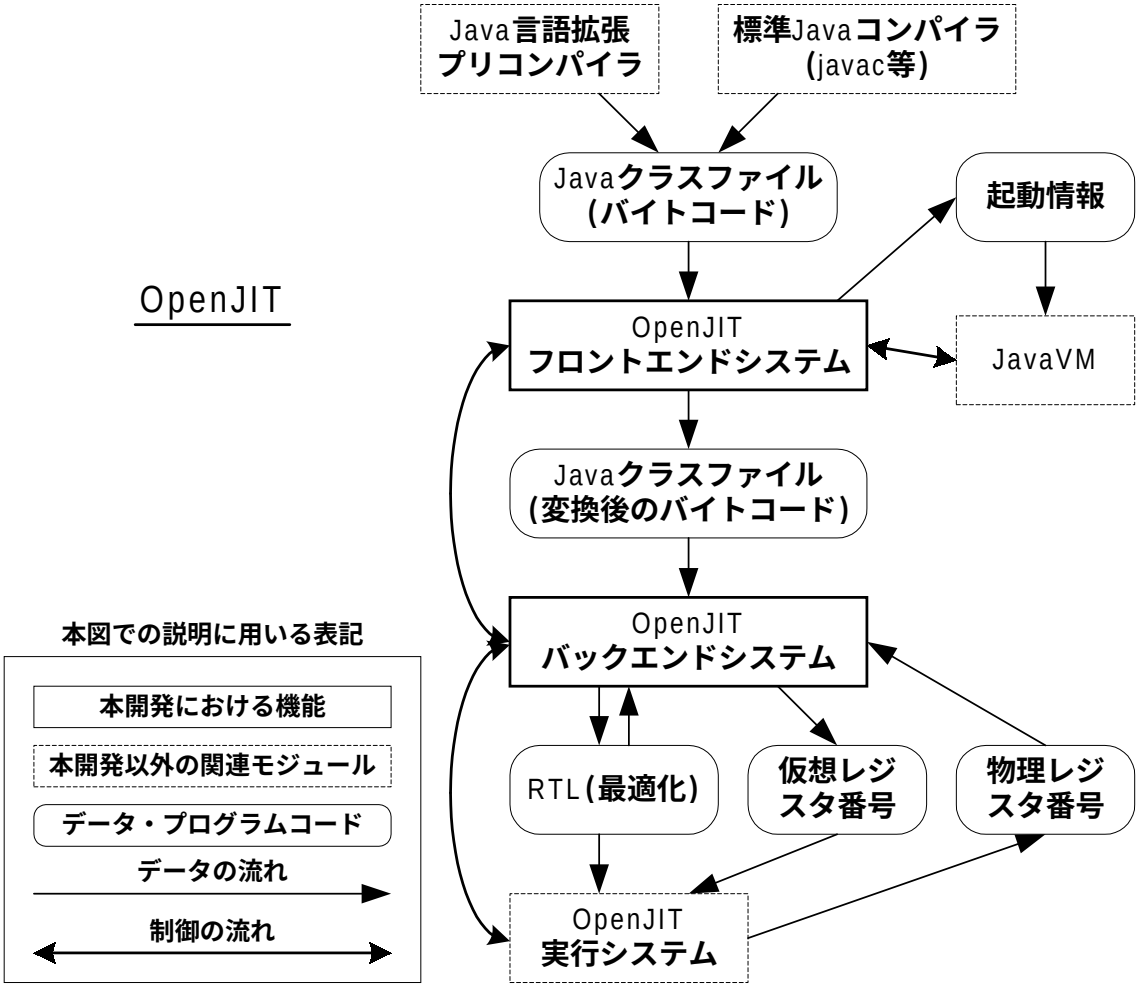

図 3.3: OpenJIT コンパイラシステムと他システムとの関連

(3)　拡張性，保守性，信頼性

本システムの拡張性，保守性，信頼性における指針およびその利点を以下に示す．

- 本システムは Open Compiler 技術に基づき，コンパイラをオブジェクト指向設計でモジュール化することにより，ユーザによる変更を可能にしている．さらに，コンパイラメタコードを記述することにより，コンパイル時にメタ計算を行ない，言語の拡張やアプリケーション固有の最適化を行うなどの拡張性を持たせることが可能になっている．

- オブジェクト指向設計に基づき，各機能をモジュール化することによって，アーキテクチャ依存部分を限定することが可能となるため，移植性が高まる．

- オブジェクト指向設計に基づき，各機能をモジュール化することによって，機能の変更・拡張のような場合にはそのモジュールのみを更新すればよいため，システムの保守が容易である．

- Java 言語を用いることにより，Java 言語自身が持つセキュリティ機能により，不正なプログラムは実行前に検出される．また，ユーザによるコンパイラメタコードの記述ミスなど，なんらかの原因によりコンパイルが失敗した場合には，インタプリタに制御を戻すことが可能である．

(4)　設計方法，文書化，設計手順，変更方法

特になし.

第 4 章

システム構成

(1)　ハードウェア構成

本システムは，以下の構成のハードウェア環境で動作する:

- Sun Ultra 60 (UltraSPARC-II×2)
- メモリ量: 256MB
- ハードディスク容量: 4GB

(2)　ソフトウェア構成

本システムは，以下の構成のソフトウェア環境で動作する:

- OS: Sun Solaris 2.6
- Java 実行環境: JDK 1.1.6

(3)　他システムとの関連

本システムは JDK からは次のように起動される．

JDK には JIT コンパイラのための API が用意されておりこれを利用する．この API を用いると，OpenJIT システムはダイナミック・リンク・ライブラリという形で JDK にアタッチされる．そして JIT コンパイラのための初期化ルーチンが呼び出される．JDK のシステムには JIT コンパイラのためにいくつかのフック関数が用意されており，初期化ルーチンではこれらのフック関数を設定する．この設定により，新しくクラスがロードされるときに，OpenJIT システムに制御が移るようになる．

新しくクラスがロードされる時，OpenJIT システムはロードしたクラスの中で定義されているメソッドのうち，native method や abstract method といったコンパイルできないものを除いたメソッドについて，メソッドが起動される時にコンパイルを行なうようにメソッドの起動関数 (invoker) を設定し直す．invoker の再設定を行なうかどうかによって，コンパイルを行なうクラスやメソッドを制限することができる．

Invoker を OpenJIT システムの起動関数に再設定することで，メソッドが起動されるとき OpenJIT システムが起動する．OpenJIT システムは起動されたメソッドをコンパイルした後，今度は invoker を直接ネイティブコードが起動するように設定し直した後，コンパイルされたネイティブコードへ制御を渡す．コンパイルされたネイティブコードはメモリ中に保存され，次回メソッドが起動された時は直接ネイティブコードが実行されるようになる．

コンパイルされたネイティブコードはメモリに蓄積されて行くが，JDK がクラスを解放する際に，フック関数が呼び出されて OpenJIT システムに制御が渡り，このときにネイティブコードも解放する．また，コンパイラのバグや何らかの原因によってコンパイルに失敗した場合は，invoker を元の設定に戻すことにより，その後はインタプリタで実行を続けることもできる．

以上のようにして，全体のシステムは動作する．このようにコンパイルの単位はメソッドであり，メソッドが始めて起動するときにだけコンパイルされ，無駄がない．

第 5 章

機能仕様

5.1 機能概要

本システムは OpenJIT フロントエンドシステムと OpenJIT バックエンドシステムから構成される．これらの機能概要を以下に示す.

5.1.1 OpenJIT フロントエンドシステム

OpenJIT フロントエンドシステムでは，基本的に与えられたバイトコードから，最適化および拡張を施したバイトコードへの変換を行う．クラスファイルに内在する Java のバイトコードを入力とし，高レベルな最適化およびプログラム変換を施して，再びバイトコードを出力する．

OpenJIT コンパイラ基盤機能は，フロントエンド系処理全体の抽象フレームワークであり，OpenJIT フロントシステムの各機能のインタフェースを定義する．このインタフェースに沿って具体的な対象アーキテクチャに応じたモジュールを記述することにより，様々な実行環境に対応することが可能となる．

OpenJIT バイトコードディスコンパイラ機能は，与えられたバイトコード列をフロー解析して，逆変換することにより，AST を得る．この際には，与えられたバイトコード列から，元のソースプログラムから生成されるコントロールグラフのリカバリを行う技術を開発する．

同時に，OpenJIT クラスファイルアノテーション解析機能により，このクラスファイルのアトリビュート領域に何らかのアノテーションが付記されていたときに，その情報を得る．たとえば，バイトコードへコンパイルしたときの高レベルな解析情報が，クラスファイルに付記されていることが考えられる．特に重要なのは，クラスファイル自身では得ることが難しいグローバルな解析情報であり，具体的には各コールサイトにおけるディスパッチ可能なクラスが挙げられる．この情報は，AST 上の付加情報として用いられる．

次に，得られた AST に対し，OpenJIT 最適化機能によって，最適化が施される．最適化に必要な情報は，OpenJIT フローグラフ構築機能，OpenJIT フローグラフ解析機能により抽出される．最適化時のプログラム変換は，OpenJIT プログラム変換機能が司って実施され，変換後のバイトコードがバックエンドシステムに出力される．

5.1.2 OpenJIT バックエンドシステム

OpenJIT フロントエンドシステムによって最適化されたバイトコード列に対し，OpenJIT バックエンドシステムは以下の技術を用いて，さらなる最適化処理を行い，ネイティブコードを出力する．

OpenJIT ネイティブコード変換機能は，バックエンド系処理全体の抽象フレームワークであり，OpenJIT バックエンドシステムの各機能のインタフェースを定義する．このインタフェースに沿って具体的なプロセッサに応じたクラスでモジュールを記述することにより，様々なプロセッサに対応することが可能となる．

OpenJIT 中間コード変換機能によって，バイトコード列からスタックオペランドを使った中間言語へと変換を行なう．バイトコードの命令を解析して分類することにより，単純な命令列に展開を行う．

得られた命令列に対し，OpenJIT RTL 変換機能は，このスタックオペランドを使った中間言語からレジスタを使った中間言語 (RTL) へ変換する．バイトコードの制御の流れを解析し，命令列を基本ブロックに分割する．バイトコードの各命令の実行時のスタックの深さを計算することで，スタックオペランドから無限個数あると仮定した仮想的なレジスタオペランドに変換する．

次に，OpenJIT Peephole 最適化機能によって，RTL の命令列の中から冗長な命令を取り除く最適化を行ない，最適化された RTL が出力され，最後に OpenJIT SPARC プロセッサコード出力モジュールにより，SPARC プロセッサのネイティブコードが出力される．OpenJIT SPARC プロセッサモジュールは，ネイティブコード生成時のレジスタ割り付けのために OpenJIT レジスタ割付機能を利用する．出力されたネイティブコードは，JavaVM によって呼び出され実行されるが，その際に OpenJIT ランタイムモジュールを補助的に呼び出す．

5.2　機能構成

本システムの機能構成を図 5.1に示す.

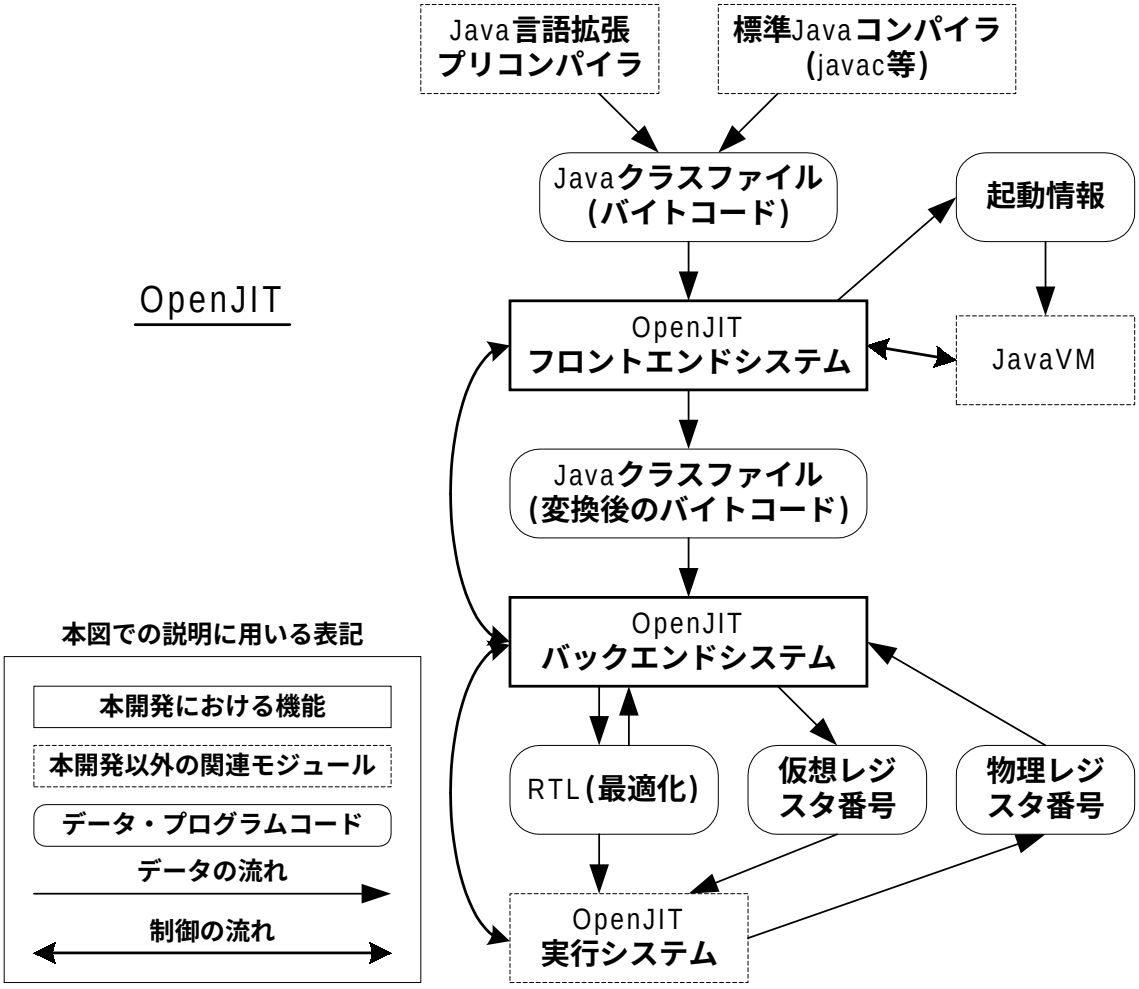

図 5.1: OpenJIT コンパイラシステムの機能構成

5.3　機能ブロック間の相互関係

本システムの機能ブロック間の相互関係を同じく図5.1に示す.

第 6 章

OpenJIT フロントエンドシステム

6.1 機能概要

OpenJIT フロントエンドシステムでは，基本的に与えられたバイトコードから，最適化および拡張を施したバイトコードへの変換を行う．クラスファイルに内在する Java のバイトコードを入力とし，高レベルな最適化およびプログラム変換を施して，再びバイトコードを出力する．

OpenJIT コンパイラ基盤機能は，フロントエンド系処理全体の抽象フレームワークであり，OpenJIT フロントシステムの各機能のインタフェースを定義する．このインタフェースに沿って具体的な対象アーキテクチャに応じたモジュールを記述することにより，様々な実行環境に対応することが可能となる．

OpenJIT バイトコードディスコンパイラ機能は，与えられたバイトコード列をフロー解析して，逆変換することにより，AST を得る．この際には，与えられたバイトコード列から，元のソースプログラムから生成されるコントロールグラフのリカバリを行う技術を開発する．

同時に，OpenJIT クラスファイルアノテーション解析機能により，このクラスファイルのアトリビュート領域に何らかのアノテーションが付記されていたときに，その情報を得る．たとえば，バイトコードへコンパイルしたときの高レベルな解析情報が，クラスファイルに付記されていることが考えられる．特に重要なのは，クラスファイル自身では得ることが難しいグローバルな解析情報であり，具体的には各コールサイトにおけるディスパッチ可能なクラスが挙げられる．この情報は，AST 上の付加情報として用いられる．

次に，得られた AST に対し，OpenJIT 最適化機能によって，最適化が施される．

最適化に必要な情報は，OpenJIT フローグラフ構築機能，OpenJIT フローグラフ解析機能により抽出される．最適化時のプログラム変換は，OpenJIT プログラム変換機能が司って実施され，変換後のバイトコードがバックエンドシステムに出力される．

6.2 機能構成及び他の機能ブロックとの関係

OpenJIT フロントエンドシステムの機能構成と他の機能ブロックとの関係を図6.1に示す.

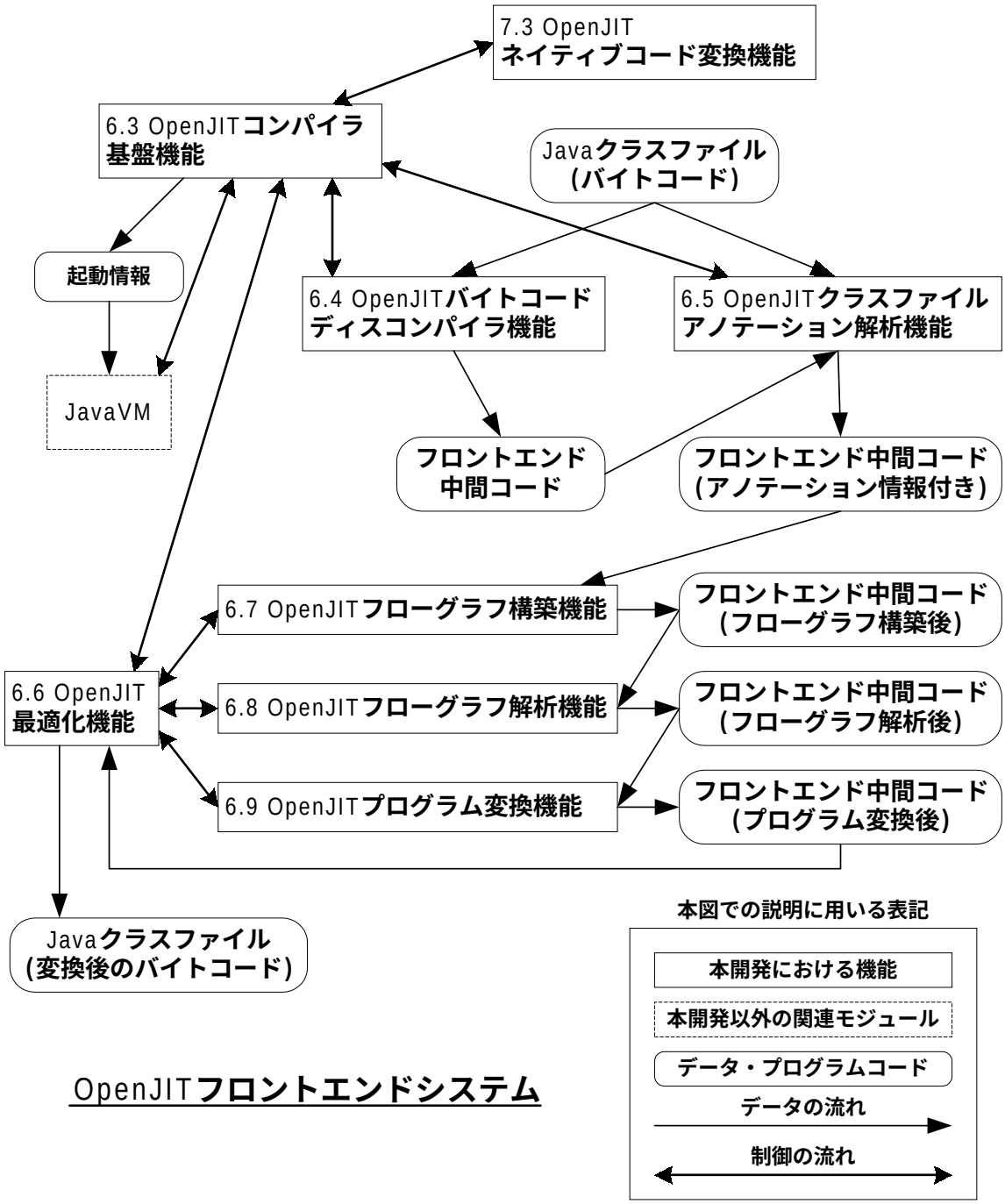

図 6.1: OpenJIT フロントエンドシステム

6.3 OpenJIT コンパイラ基盤機能

(1) 機能説明

本機能ではOpenJIT 全体の基本動作を司る.

Sun の JDK においては, Java Native Code API(Application Programmer's Interface) というコンパイラに対するインタフェースが用意されている. この API は JVM のインタプリタにネイティブコード生成を組み込むために用意されたものである. 今回開発する OpenJIT コンパイラでは, この API に基づくことにより JDK に準拠の VM に OpenJIT コンパイラを組み込むことができる. この JIT コンパイラは JVM から必要なときに読み込まれ動作する.

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT コンパイラ基盤機能の構成と他の機能ブロックとの関係を図 6.2に示す.

(3) 入力データ

特になし.

(4) 出力データ

起動情報.

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

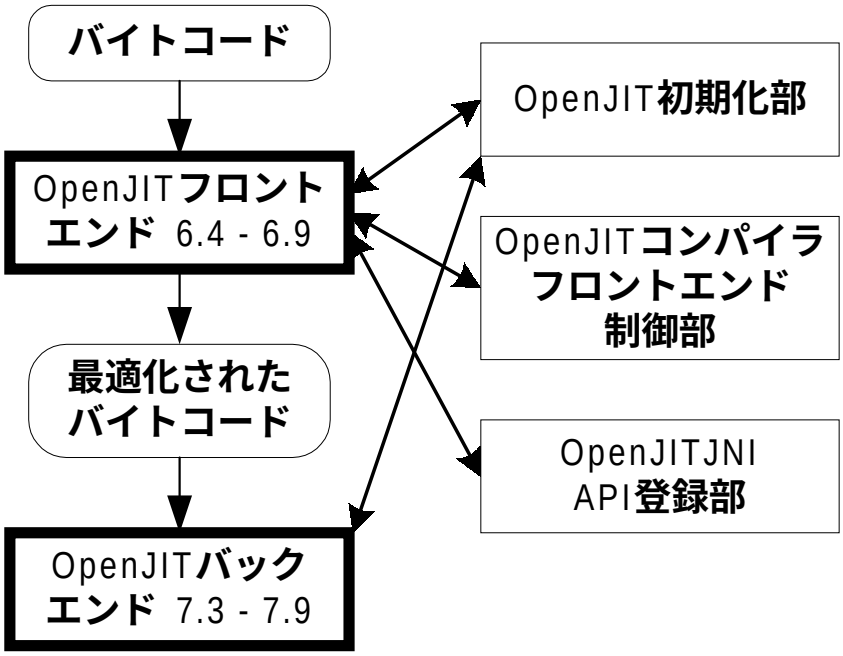

図 6.2: OpenJIT コンパイラ基盤機能

6.3.1 OpenJIT 初期化部

(1) 機能説明

OpenJIT 初期化部は，OpenJIT フロントエンドシステムの各機能，OpenJIT バックエンドシステムの各機能の初期化を指示し，OpenJIT システム全体の初期化を行う．

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.2に示す通りである．

(3) 入力データ

特になし．

(4) 出力データ

特になし．

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

6.3.2 OpenJIT コンパイラフロントエンド制御部

(1) 機能説明

OpenJIT フロントエンドシステムの OpenJIT バイトコードディスコンパイラ機能, OpenJIT クラスファイルアノテーション解析機能, OpenJIT 最適化機能を必要に応じて，起動および制御することで，フロントエンド系の処理を制御する.

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.2に示す通りである.

(3) 入力データ

特になし.

(4) 出力データ

特になし.

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

6.3.3 OpenJIT JNI API 登録部

(1) 機能説明

OpenJIT バックエンドシステム用の JNI (Java Native Interface) の API を登録する. これにより, JavaVM に対して付加される JIT 用に JDK が用意した API を OpenJIT システム内から利用して, 任意のメソッドを JIT コンパイルすることが可能となる.

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.2に示す通りである.

(3) 入力データ

特になし.

(4) 出力データ

特になし.

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

6.4 OpenJIT バイトコードディスコンパイラ機能

(1) 機能説明

OpenJIT バイトコードディスコンパイラ機能は，Java のクラスファイルのそれぞれのバイトコードを，いわゆる discompiler 技術により，バイトコードレベルからコントロールフローグラフ，AST(抽象構文木) を含む抽象化レベルのプログラム表現を復元し，以後の OpenJIT の各モジュールの操作の対象として利用可能にする処理を行う.

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT バイトコードディスコンパイラ機能の構成と他の機能ブロックとの関係を図 6.3に示す.

(3) 入力データ

バイトコード

(4) 出力データ

AST，コントロールフローグラフ

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

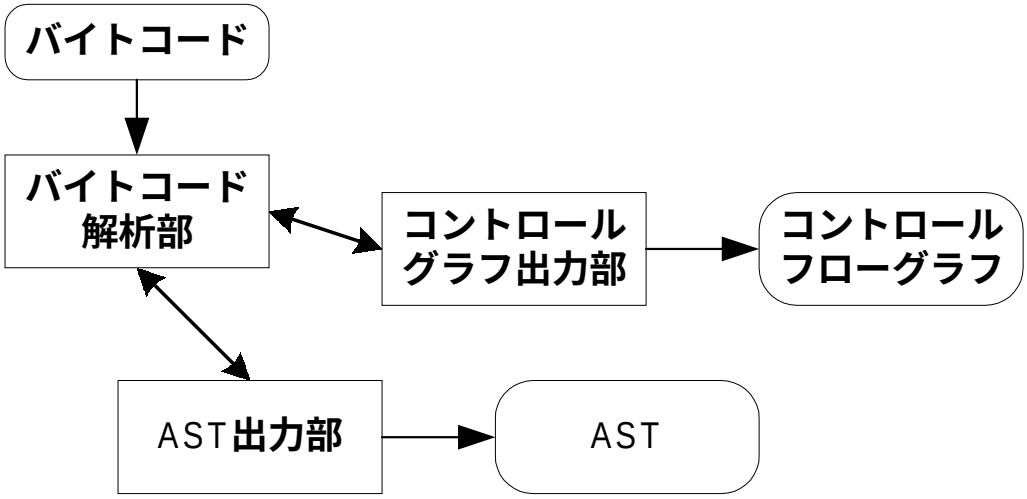

図 6.3: OpenJIT バイトコードディスコンパイラ機能

6.4.1　バイトコード解析部

(1)　機能説明

バイトコードを入力とし，コントロールグラフ出力部及び AST 出力部での解析に適した内部表現 (VMInstruction オブジェクトの配列) に変換した後に，コントロールグラフの出力と AST の出力を指示する。

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.3に示す通りである.

(3)　入力データ

バイトコード.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.4.2　コントロールグラフ出力部

(1)　機能説明

バイトコード解析部に入力されたバイトコードから得られたバイトコードの内部表現 (VMInstruction オブジェクトの配列) を解析し，そのバイトコードの表すコントロールフローグラフを出力する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.3に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

コントロールフローグラフ.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.4.3 AST 出力部

(1) 機能説明

バイトコード解析部に入力されたバイトコードから得られたバイトコードの内部表現 (VMInstruction オブジェクトの配列) を，いわゆるディスコンパイラ技術を用いて解析し，同じ意味を表す Java 言語の AST に変換した後に出力する.

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.3に示す通りである.

(3) 入力データ

特になし.

(4) 出力データ

AST.

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

6.5 OpenJIT クラスファイルアノテーション解析機能

(1) 機能説明

OpenJIT クラスファイルアノテーション解析機能は，アノテーション情報を解析し， OpenJIT ディスコンパイラ機能が生成したプログラムグラフ (AST) に対して，コンパイル時に適切な拡張された OpenJIT のメタクラスを起動できるようにする.

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT クラスファイルアノテーション解析機能の構成と他の機能ブロックとの関係を図 6.4に示す.

(3) 入力データ

アノテーションを付記したクラスファイル, AST

(4) 出力データ

AST に対する付加データ

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

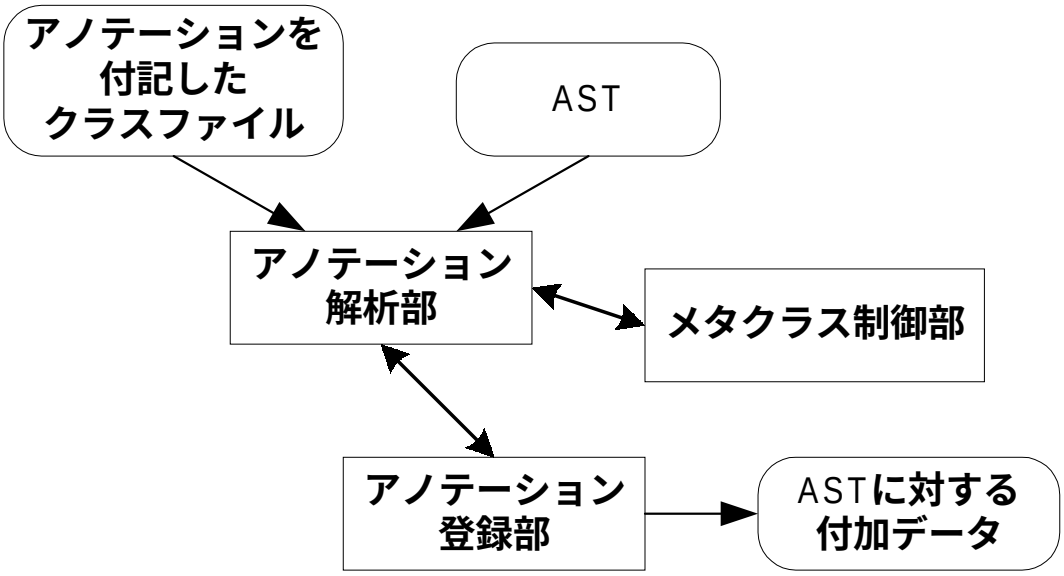

図 6.4: OpenJIT クラスファイルアノテーション解析機能

6.5.1 アノテーション解析部

(1) 機能説明

アノテーションを付記したクラスファイルおよびASTを入力とし，クラスファイルに付加されたアノテーションを解析して，必要に応じてメタクラス制御部，アノテーション登録部を制御する.

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図6.4に示す通りである.

(3) 入力データ

アノテーションを付記したクラスファイル，AST.

(4) 出力データ

特になし.

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

6.5.2　アノテーション登録部

(1)　機能説明

アノテーション解析部から呼び出され，アノテーション解析部で解析されたアノテーションを AST に対する付加データとして登録して，追加情報を付加した AST を出力する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.4に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

AST に対する付加データ.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.5.3　メタクラス制御部

(1)　機能説明

アノテーション解析部から呼び出され，アノテーション解析部が解析したアノテーション情報にしたがって，ASTを処理するための適切なメタクラスが起動されるように制御を行う.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図6.4に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.6 OpenJIT 最適化機能

(1) 機能説明

OpenJIT 最適化機能は，OpenJIT フローグラフ構築機能，OpenJIT フローグラフ解析機能，および OpenJIT プログラム変換機能を用い，プログラム最適化を行なう．OpenJIT コンパイラには，標準的なコンパイラの最適化を含む最適化ライブラリ構築のためのサポートが準備される．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT 最適化機能の構成と他の機能ブロックとの関係を図 6.5に示す．

(3) 入力データ

(最適化前の) バイトコード，AST，コントロールフローグラフ

(4) 出力データ

最適化されたバイトコード

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

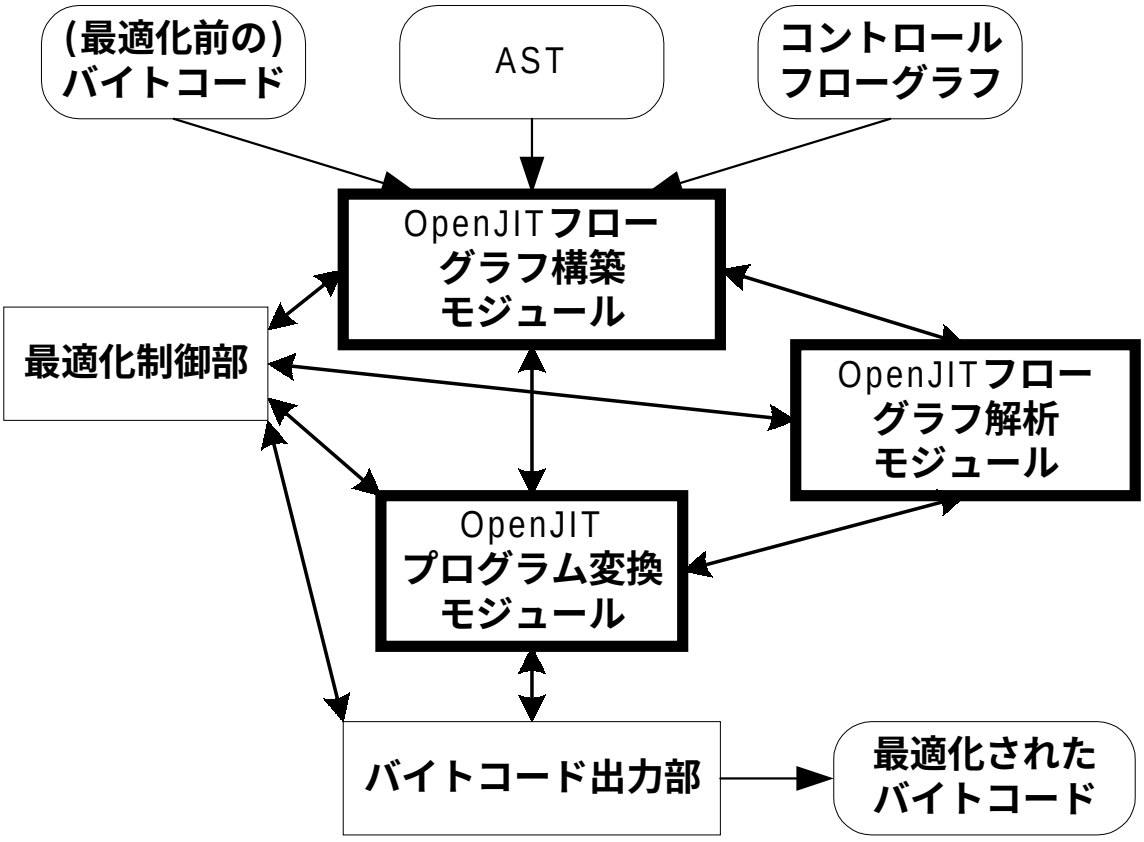

図 6.5: OpenJIT 最適化機能

6.6.1 最適化制御部

(1) 機能説明

OpenJIT フローグラフ構築機能，OpenJIT フローグラフ解析機能，OpenJIT プログラム変換機能，バイトコード出力部を制御して，入力のバイトコード，AST，コントロールフローグラフから，最適化されたバイトコードの生成を指示する．

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.5に示す通りである．

(3) 入力データ

特になし．

(4) 出力データ

特になし．

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

6.6.2 バイトコード出力部

(1) 機能説明

最適化制御部から呼び出され，OpenJIT プログラム変換機能により最適化を施された AST をコンパイルして，バイトコードを生成して，出力する．出力されたバイトコードは OpenJIT バックエンドシステムによって利用される．

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.5に示す通りである．

(3) 入力データ

特になし．

(4) 出力データ

最適化されたバイトコード．

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

6.7 OpenJIT フローグラフ構築機能

(1) 機能説明

OpenJIT フローグラフ構築機能は，AST およびコントロールフローグラフを受け取り，対応するデータフローグラフ，コントロール依存グラフ，を含むフローグラフを出力する．また，クラスファイル間のクラス階層情報情報を得られる場合は，クラスファイルの関係を読み込み，オブジェクト指向解析用のクラス階層グラフも出力する．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT フローグラフ構築機能の構成と他の機能ブロックとの関係を図6.6に示す．

(3) 入力データ

AST，コントロールフローグラフ，クラスファイル間のクラス階層情報

(4) 出力データ

データフローグラフ，コントロール依存グラフ，クラス階層グラフ

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

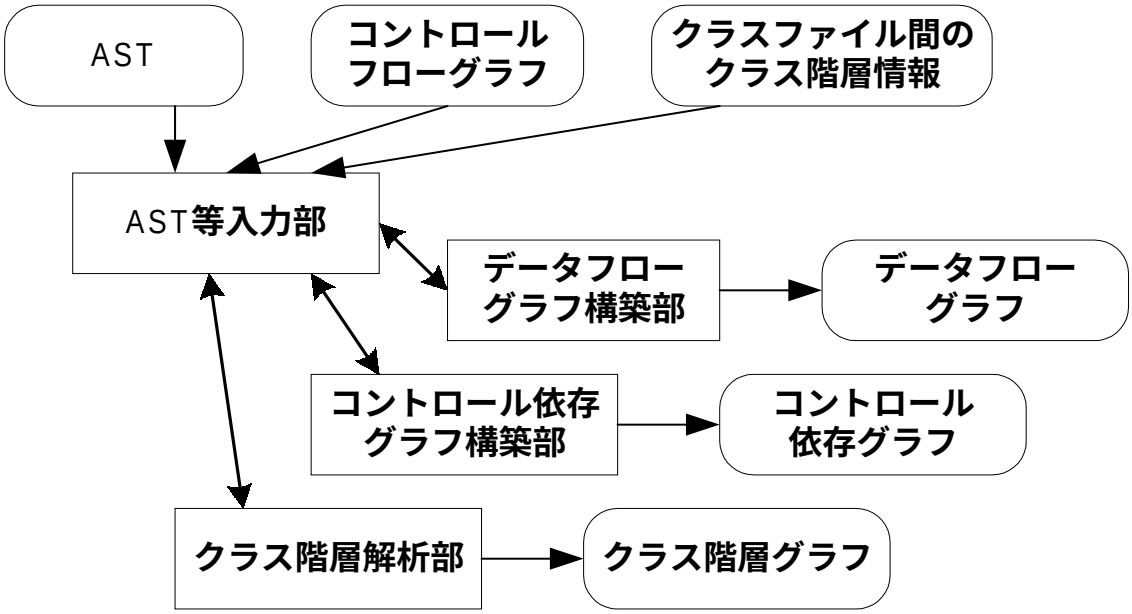

図 6.6: OpenJIT フローグラフ構築機能

6.7.1 AST 等入力部

(1) 機能説明

AST，コントロールフローグラフ，クラスファイル間のクラス階層情報を入力し，中間形式に変換して，その情報をもとにデータフローグラフ構築部，コントロール依存グラフ構築部，クラス階層解析部にそれぞれ処理を指示する．

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.6に示す通りである．

(3) 入力データ

AST，コントロールフローグラフ，クラスファイル間のクラス階層情報．

(4) 出力データ

特になし．

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

6.7.2　データフローグラフ構築部

(1)　機能説明

AST 等入力部から呼び出され，AST 等入力部に入力されたプログラム情報から，データフローグラフを構築して出力する．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.6に示す通りである．

(3)　入力データ

特になし．

(4)　出力データ

データフローグラフ．

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

6.7.3　コントロール依存グラフ構築部

(1)　機能説明

AST 等入力部から呼び出され，AST 等入力部に入力されたプログラム情報である AST から，コントロール依存グラフを構築して出力する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.6に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

コントロール依存グラフ.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.7.4　クラス階層解析部

(1)　機能説明

AST 等入力部から呼び出され，AST 等入力部に入力されたプログラム情報である AST およびクラスファイル間のクラス階層情報を用いて，クラス階層解析を行い，その情報を付加したクラス階層グラフを構築して出力する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.6に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

クラス階層グラフ.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.8 OpenJIT フローグラフ解析機能

(1) 機能説明

OpenJIT フローグラフ解析機能は，OpenJIT フローグラフ構築機能で構築されたプログラム表現のグラフに対して，グラフ上の解析を行う．基本的には，一般的なグラフのデータフロー問題として定式化され，トップダウンおよびボトムアップの解析のベースとなる汎用的なアルゴリズムをサポートする．具体的には，グラフ上のデータフロー問題，マージ，不動点検出，などの一連のアルゴリズムがメソッド群として用意される．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT フローグラフ解析機能の構成と他の機能ブロックとの関係を図6.7に示す．

(3) 入力データ

コントロールフローグラフ，データフローグラフ，コントロール依存グラフ，クラス階層グラフ

(4) 出力データ

プログラム解析結果のデータ

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

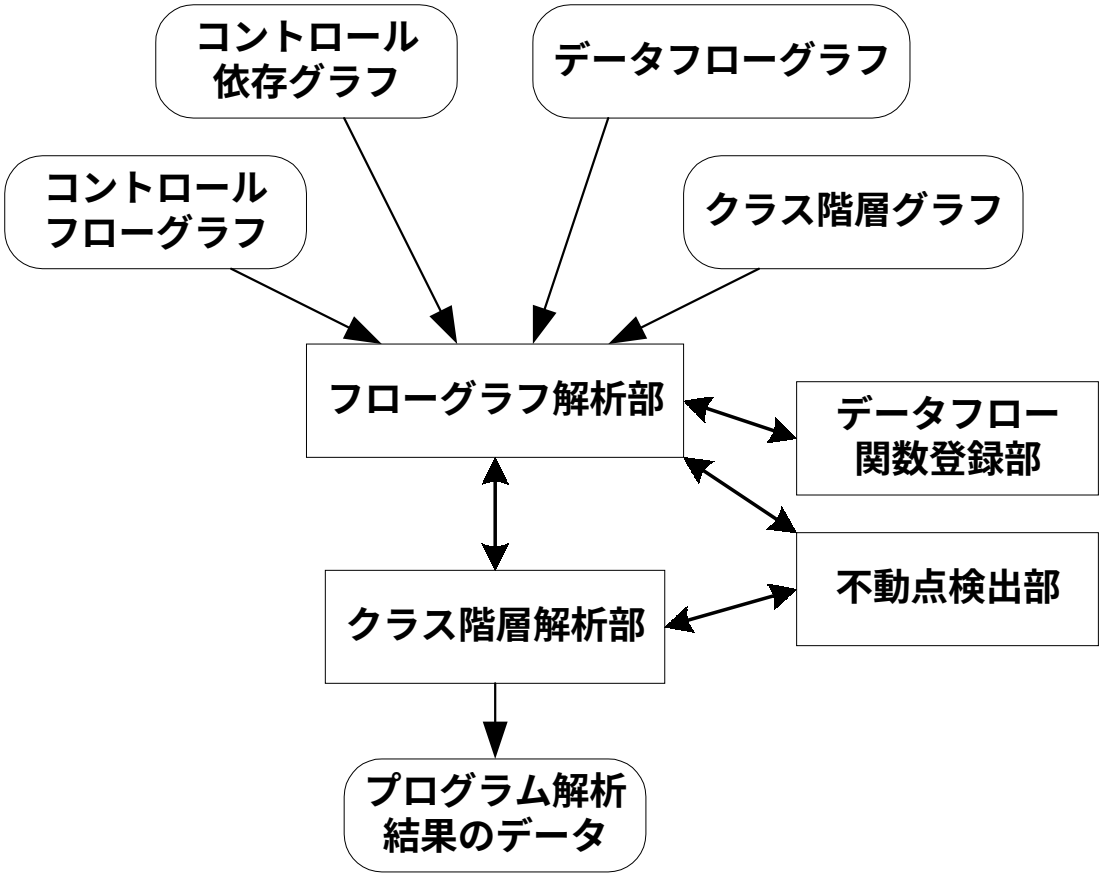

図 6.7: OpenJIT フローグラフ解析機能

6.8.1　データフロー関数登録部

(1)　機能説明

フローグラフ解析部から呼び出され，データフロー解析に用いるデータフロー関数を登録する．

グラフ上のデータフロー問題を実現するデータフロー関数群はこのサブモジュールに含まれる．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図6.7に示す通りである．

(3)　入力データ

特になし．

(4)　出力データ

特になし．

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

6.8.2　フローグラフ解析部

(1)　機能説明

コントロールフローグラフ，コントロール依存グラフ，データフローグラフ，クラス階層グラフを入力として，登録されたデータフロー関数を用いてデータフロー解析を行う．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図6.7に示す通りである．

(3)　入力データ

コントロールフローグラフ，コントロール依存グラフ，データフローグラフ，クラス階層グラフ．

(4)　出力データ

なし．

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

6.8.3　不動点検出部

(1)　機能説明

フローグラフ解析部から呼び出され，フローグラフ解析部に入力されたデータフローグラフの不動点を検出する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図6.7に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.8.4　クラス階層解析部

(1)　機能説明

フローグラフ解析部から呼び出され，フローグラフ解析部に入力されたクラス階層グラフと，データフローグラフの解析結果から，クラス階層の解析を行い，その結果を出力する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図6.7に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

プログラム解析結果のデータ.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

6.9 OpenJIT プログラム変換機能

(1) 機能説明

OpenJIT プログラム変換機能では，OpenJIT フローグラフ解析機能の結果やユーザのコンパイラのカスタマイゼーションにしたがって，プログラム変換を行う．プログラム変換のためには，AST の書き換え規則がユーザによって定義され，AST 上のパターンマッチが行われ，適用された規則に従ってプログラムの書き換えが行われる．書換え規則自身，全て Java のオブジェクトとして定義され，ユーザはあらかじめ書換え規則を定義して，OpenJIT プログラム変換機能に登録しておく．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT プログラム変換機能の構成と他の機能ブロックとの関係を図 6.8に示す．

(3) 入力データ

AST，プログラム解析結果のデータ

(4) 出力データ

変換された AST

(5) 例外条件及び制約条件

(6) 特記事項（性能，容量）

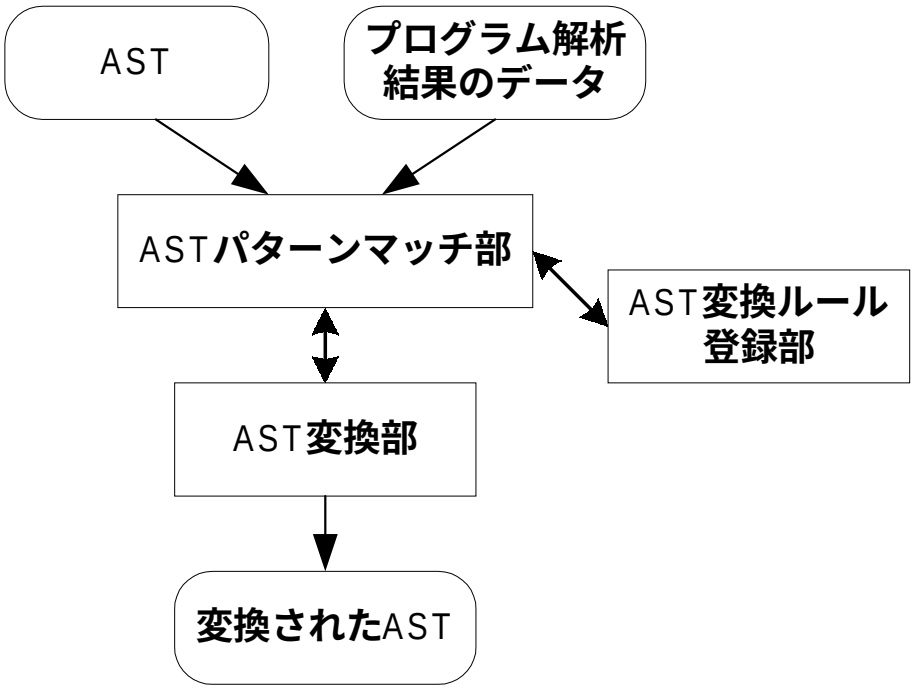

図 6.8: OpenJIT プログラム変換機能

6.9.1 AST 変換ルール登録部

(1) 機能説明

パターンマッチによる AST 上のプログラム変換を実現するために，AST パターンマッチ部で用いる AST から AST へのパターンマッチの変換ルールを登録する．

パターンマッチの変換ルールはこのサブモジュールに含まれる．

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.8に示す通りである．

(3) 入力データ

特になし．

(4) 出力データ

特になし．

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

6.9.2 AST パターンマッチ部

(1) 機能説明

AST とプログラム解析結果のデータを入力として，AST 変換ルール登録部で登録された変換ルールを用いてパターンマッチを行い，プログラムグラフの AST の置き換えを行う.

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.8に示す通りである.

(3) 入力データ

AST，プログラム解析結果のデータ.

(4) 出力データ

特になし.

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

6.9.3 AST 変換部

(1) 機能説明

パターンマッチした変換規則を用いて，AST の書き換えを行って変換された AST を出力する.

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 6.8に示す通りである.

(3) 入力データ

特になし.

(4) 出力データ

変換された AST.

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

第 7 章

OpenJIT バックエンドシステム

7.1 機能概要

OpenJIT フロントエンドシステムによって最適化されたバイトコード列に対し，OpenJIT バックエンドシステムは以下の技術を用いて，さらなる最適化処理を行い，ネイティブコードを出力する．

OpenJIT ネイティブコード変換機能はバックエンド系処理全体の抽象フレームワークであり，OpenJIT バックエンドシステムの各機能のインタフェースを定義する．このインタフェースに沿って具体的なプロセッサに応じたクラスでモジュールを記述することにより，様々なプロセッサに対応することが可能となる．

OpenJIT 中間コード変換機能によって，バイトコード列からスタックオペランドを使った中間言語へと変換を行なう．バイトコードの命令を解析して分類することにより，単純な命令列に展開を行う．

得られた命令列に対し，OpenJIT RTL 変換機能は，このスタックオペランドを使った中間言語からレジスタを使った中間言語 (RTL) へ変換する．バイトコードの制御の流れを解析し，命令列を基本ブロックに分割する．バイトコードの各命令の実行時のスタックの深さを計算することで，スタックオペランドから無限個数あると仮定した仮想的なレジスタオペランドに変換する．

次に，OpenJIT Peephole 最適化機能によって，RTL の命令列の中から冗長な命令を取り除く最適化を行ない，最適化された RTL が出力され，最後に OpenJIT SPARC プロセッサコード出力モジュールにより，SPARC プロセッサのネイティブコードが出力される．OpenJIT SPARC プロセッサモジュールは，ネイティブコード生成時のレジスタ割り付けのために OpenJIT レジスタ割付機能を利用する．出力されたネイ

ティブコードは，JavaVM によって呼び出され実行されるが，その際に OpenJIT ランタイムモジュールを補助的に呼び出す.

7.2 機能構成及び他の機能ブロックとの関係

OpenJIT フロントエンドシステムの機能構成と他の機能ブロックとの関係を図 7.1に示す.

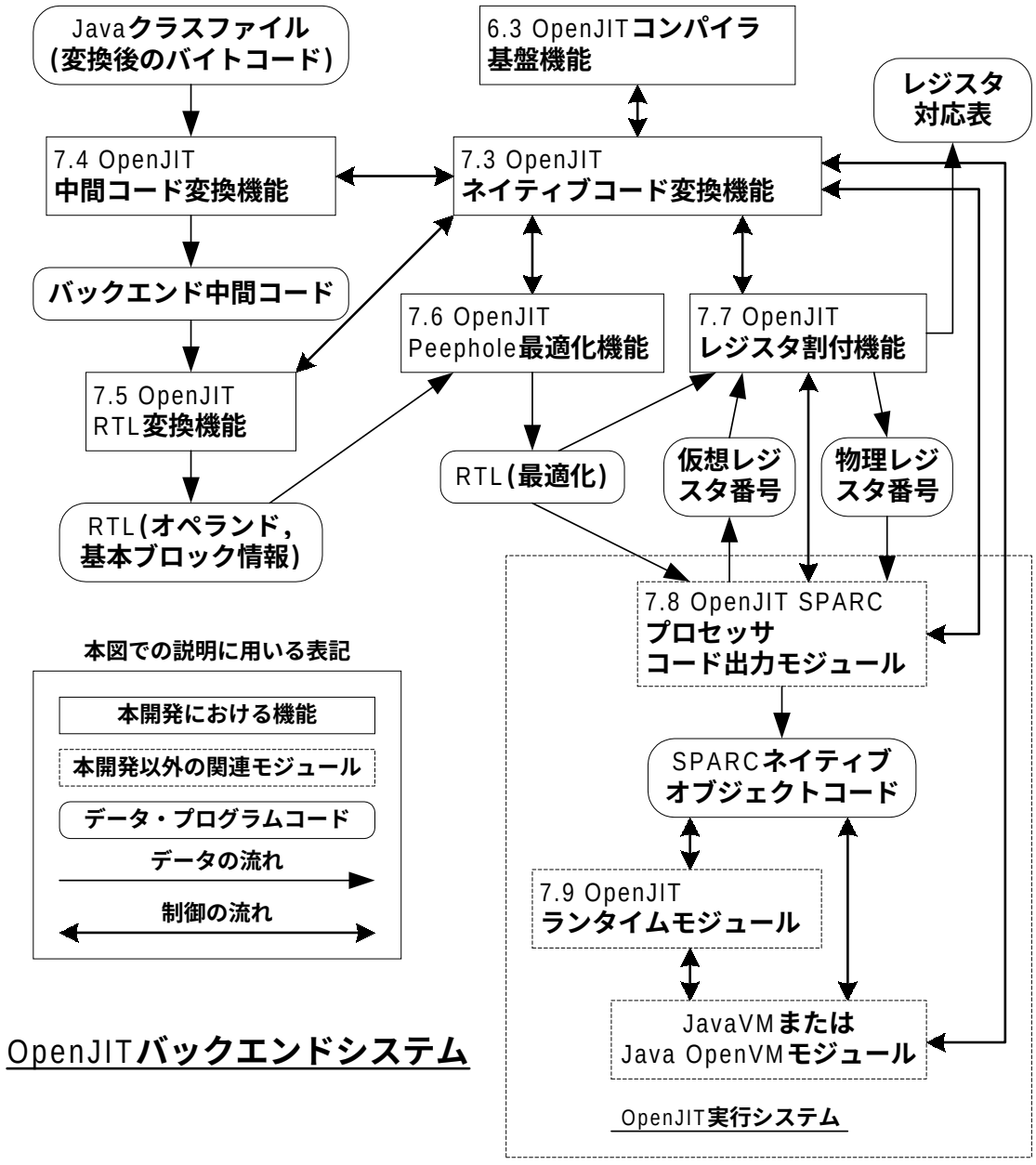

図 7.1: OpenJIT バックエンドシステム

7.3 OpenJIT ネイティブコード変換機能

(1) 機能説明

Java のバイトコードからネィティブコードを出力するための抽象フレームワークである．実際は，このクラスを具体的なプロセッサに応じたクラスで特化することによって，実際のコード出力機能を定義する．それぞれのバイトコードと，プログラムの各種グラフ，およびフロントエンドシステムのプログラム解析・変換の結果を用いて，ネィティブコードへの変換を行なう．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT ネイティブコード変換機能の構成と他の機能ブロックとの関係を図 7.2に示す．

(3) 入力データ

フロントエンド変換後のバイトコード

(4) 出力データ

SPARC プロセッサのネイティブコード

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

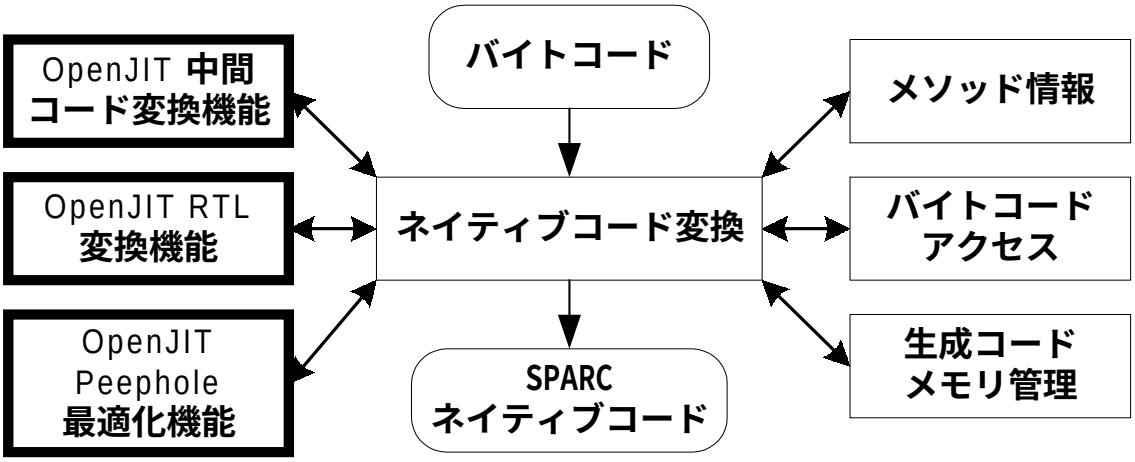

図 7.2: OpenJIT ネイティブコード変換機能

7.3.1 ネイティブコード変換

(1) 機能説明

バイトコードを入力とし，SPARC ネイティブコードを出力する OpenJIT 中間コード変換機能，OpenJIT RTL 変換機能，OpenJIT Peephole 最適化機能を制御する．

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.2に示す通りである．

(3) 入力データ

バイトコード

(4) 出力データ

SPARC ネイティブコード

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

7.3.2　メソッド情報

(1)　機能説明

メソッドに関する JDK の内部構造を Java のデータ構造に変換する.
constant pool のインデックス値をもとに，constant pool に関する，定数値や定数の格納アドレス，型の情報，クラス名，メソッド名，フィールド名を取り出す.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.2に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

7.3.3　バイトコードアクセス

(1)　機能説明

JDK の内部構造であるバイトコードを読み取る.
バイトコード列から，符号なし１バイトデータ，符号無し２バイトデータ，符号付き１バイトデータ，符号付き２バイトデータ，符号付き４バイトデータを読み出す.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.2に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

7.3.4　生成コードメモリ管理

(1)　機能説明

生成するネイティブコードの領域の確保，生成したネイティブコードの領域の再割り当てを行い，その領域への生成したネイティブコードの書き込み，読み出しを行う．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.2に示す通りである．

(3)　入力データ

特になし．

(4)　出力データ

特になし．

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

7.4 OpenJIT 中間コード変換機能

(1) 機能説明

フロントエンドシステムの出力であるバイトコードを入力とする．バイトコードの各命令をグループに分別し，より単純な中間言語に変換を行なう．メソッド呼び出しのバイトコード命令について，メソッドの引数の数や型の解析を行い，中間言語列に展開する．この中間言語のオペランドはスタックで与えられる．また，Java 特有な命令列パターンを検出し，単純な中間言語に置き換える最適化を含めて行う．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT 中間コード変換機能の構成と他の機能ブロックとの関係を図 7.3に示す．

(3) 入力データ

バイトコード

(4) 出力データ

中間言語

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

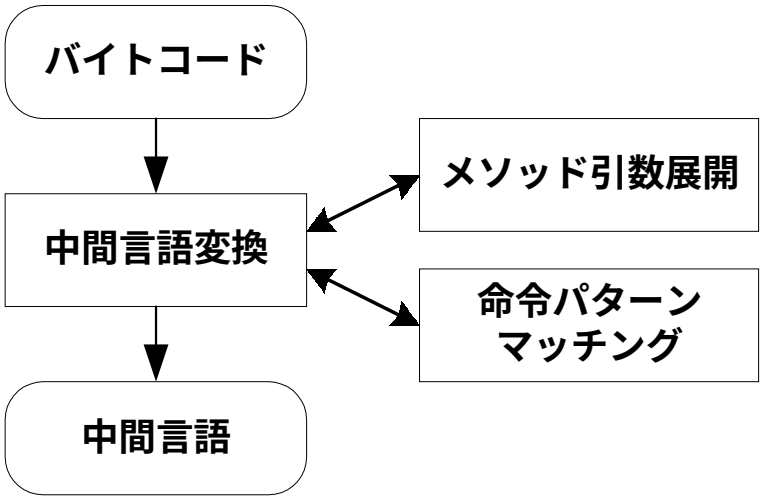

図 7.3: OpenJIT 中間コード変換機能

7.4.1 中間言語変換

(1) 機能説明

バイトコードを入力として，バイトコードの種別を調べ，constant pool にオペランドを持つ命令の場合，constant pool の値を展開して中間言語のオペランドにし，その種別に応じた中間言語を出力する.

(2) 機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.3に示す通りである.

(3) 入力データ

バイトコード

(4) 出力データ

中間言語

(5) 例外条件及び制約条件

特になし.

(6) 特記事項（性能，容量）

特になし.

7.4.2　メソッド引数展開

(1)　機能説明

constant pool からメソッドの signature を読み出し，これにしたがってメソッドの呼び出しを行うときの，引数を設定する中間言語に展開する．また，メソッドの返り値の型に従って，Java のスタックにメソッドの返り値を設定する中間言語も展開する．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.3に示す通りである．

(3)　入力データ

特になし．

(4)　出力データ

特になし．

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

7.4.3　命令パターンマッチング

(1)　機能説明

特定のバイトコードパターンに対し最適化した中間言語に変換する.
boolean 値の否定を行うバイトコードのパターン，long 値の比較・分岐を行うバイトコードのパターン，float や double 値の比較・分岐を行うバイトコードのパターンにマッチするバイトコードを見つけ，最適化した中間言語に変換する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.3に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

7.5 OpenJIT RTL 変換機能

(1) 機能説明

OpenJIT 中間コード変換機能の生成結果を入力とし，スタックオペランドを使った中間言語からレジスタを使った中間言語，RTL(Register Transfer Language) に変換を行なう．中間言語列を基本ブロックに分割し，実行の制御の流れを解析することにより，スタックマシンコードからオペランドレジスタコードへの変換を行なう．OpenJIT では，無限資源のレジスタがあるとみなして RTL への変換を行なう．また，オペランドのうち型が未解決のものの型を決定する．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT RTL 変換機能の構成と他の機能ブロックとの関係を図 7.4に示す．

(3) 入力データ

中間言語

(4) 出力データ

RTL，基本ブロック情報

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

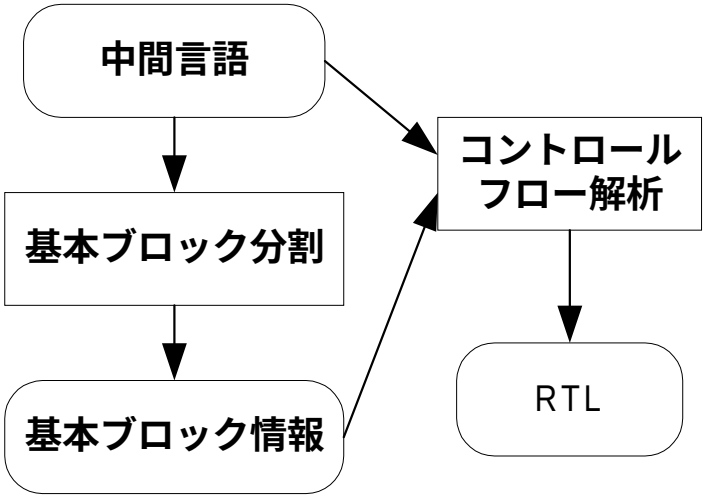

図 7.4: OpenJIT RTL 変換機能

7.5.1　基本ブロック分割

(1)　機能説明

中間言語を入力とし，中間言語の分岐命令と分岐先のアドレスを解析し，中間言語列を基本ブロック単位に分割して，基本ブロックの開始アドレスと終了アドレスを持つ基本ブロック情報を出力する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.4に示す通りである.

(3)　入力データ

中間言語

(4)　出力データ

基本ブロック情報

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

7.5.2　コントロールフロー解析

(1)　機能説明

中間言語，基本ブロック情報を入力として，中間言語の先頭から処理の流れを追跡して，基本ブロックの先頭での Java のスタックの状態を解析する．このスタックの状態をもとにして，中間言語のオペランドを変換し，RTL を出力する．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.4に示す通りである．

(3)　入力データ

中間言語，基本ブロック情報

(4)　出力データ

RTL

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

7.6 OpenJIT Peephole 最適化機能

(1) 機能説明

OpenJIT RTL 変換機能の生成した RTL を入力として，RTL に対して Peephole 最適化を施す．Peephole 最適化としては，通常行われる redundant load/store elimination を行う．また，Java 固有の Peephole 最適化も行なわれる．Java に特有な配列のインデックスの境界チェックを取り除く最適化も行なう．このモジュールは冗長な命令を取り除いて最適化された RTL を出力する．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT Peephole 最適化機能の構成と他の機能ブロックとの関係を図 7.5に示す．

(3) 入力データ

RTL，基本ブロック情報

(4) 出力データ

最適化された RTL

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

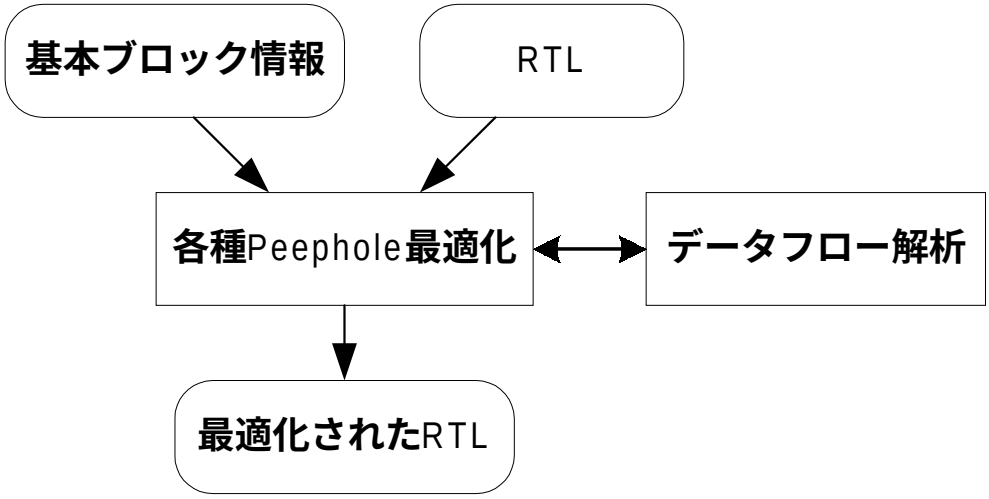

図 7.5: OpenJIT Peephole 最適化機能

7.6.1　データフロー解析

(1)　機能説明

基本ブロック内のRTL列において，データがどの命令で設定され，どの命令で利用されるかを解析する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図7.5に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

7.6.2　各種 Peephole 最適化

(1)　機能説明

基本ブロック情報と RTL を入力として，基本ブロックを単位としたデータフロー解析結果を元に Peephole 最適化を行い冗長なＲＴＬを除去する．また，constant folding 最適化も行う．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.5に示す通りである．

(3)　入力データ

基本ブロック情報，RTL

(4)　出力データ

最適化された RTL

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

7.7 OpenJIT レジスタ割付機能

(1) 機能説明

ネイティブコード生成の際，実際のプロセッサレジスタへの割付を行なう．レジスタ割付アルゴリズムを適用し，実際のプロセッサレジスタに対して割付を行なう．物理レジスタの数が足りない場合は，一時レジスタを割り付け，スピル / フィルコードを生成する．

(2) 機能構成及び他の機能ブロックとの関係

OpenJIT レジスタ割付機能の構成と他の機能ブロックとの関係を図 7.6に示す．

(3) 入力データ

仮想レジスタ番号

(4) 出力データ

物理レジスタ番号，スピル / フィルコード

(5) 例外条件及び制約条件

特になし．

(6) 特記事項（性能，容量）

特になし．

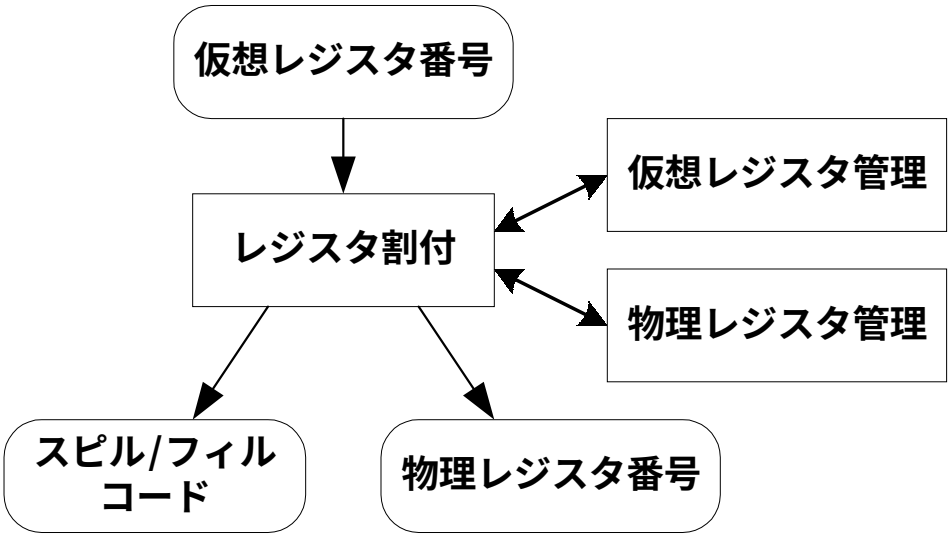

図 7.6: OpenJIT レジスタ割付機能

7.7.1　仮想レジスタ管理

(1)　機能説明

仮想レジスタの型，物理レジスタ番号との対応，スピル／フィルを行うときのメモリアドレスを指定するベースレジスタの番号やオフセットの管理を行う．

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.6に示す通りである．

(3)　入力データ

特になし．

(4)　出力データ

特になし．

(5)　例外条件及び制約条件

特になし．

(6)　特記事項（性能，容量）

特になし．

7.7.2　物理レジスタ管理

(1)　機能説明

利用可能な物理レジスタの番号を管理し，割り付けられた仮想レジスタとの対応や，一時レジスタの物理レジスタへの割り付けや解放を司る.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.6に示す通りである.

(3)　入力データ

特になし.

(4)　出力データ

特になし.

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

7.7.3　レジスタ割付

(1)　機能説明

与えられた仮想レジスタ番号に対して，物理レジスタを割り付け，物理レジスタ番号を返す.
割り付けできないときは一時レジスタを割り付け，スピル／フィルコードを生成する.

(2)　機能構成及び他の機能ブロックとの関係

本小機能の他の機能ブロックとの関係は図 7.6に示す通りである.

(3)　入力データ

仮想レジスタ番号

(4)　出力データ

スピル / フィルコード，物理レジスタ番号

(5)　例外条件及び制約条件

特になし.

(6)　特記事項（性能，容量）

特になし.

7.8 OpenJIT SPARC プロセッサコード出力モジュール

(注: 本開発においては，本モジュールの開発は富士通・大学側の負担で行われ，契約の対象とはしない．)

7.9 OpenJIT ランタイムモジュール

(注: 本開発においては，本モジュールの開発は富士通・大学側の負担で行われ，契約の対象とはしない．)

第 8 章

入出力仕様

8.1 OpenJIT フロントエンドシステム

8.1.1 概要

OpenJIT フロントエンドシステムにおける入出力データの仕様を以下に記述する.

8.1.2　入出力データ仕様

(1)　OpenJIT コンパイラ基盤機能

OpenJIT コンパイラ基盤機能を構成するサブプログラムを図8.1に示す．これらサブプログラム間で用いられる入出力データはない．

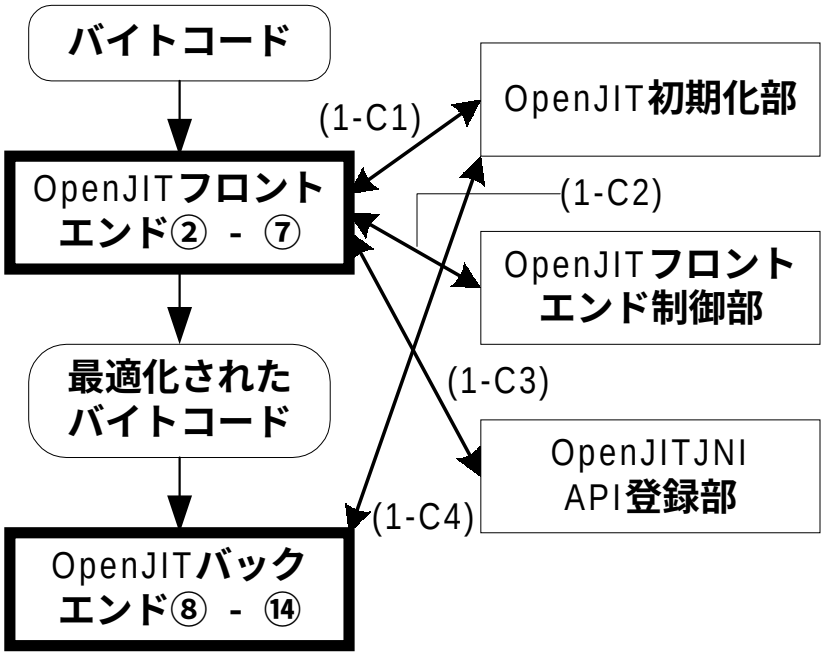

図 8.1: OpenJIT コンパイラ基盤機能

(2) OpenJIT バイトコードディスコンパイラ機能

OpenJIT バイトコードディスコンパイラ機能を構成するサブプログラムを図8.2に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(2-D1) バイトコード

クラスファイルからの変換前のバイトコード．

(2-D2) コントロールフローグラフ

バイトコードのディスコンパイルして得られたコントロールフローグラフ．

(2-D3) AST

バイトコードディスコンパイラが出力したプログラムグラフの AST．

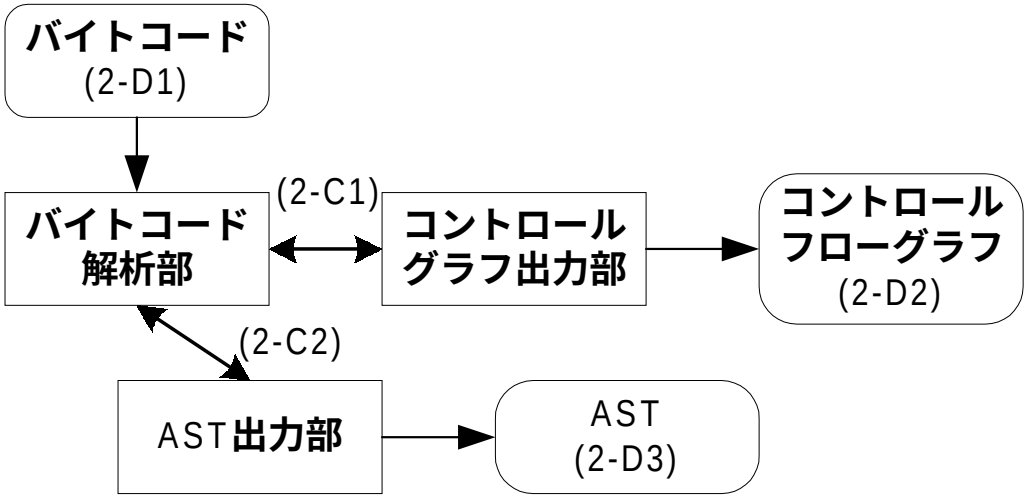

図 8.2: OpenJIT バイトコードディスコンパイラ機能

(3) OpenJIT クラスファイルアノテーション解析機能

OpenJIT クラスファイルアノテーション解析機能を構成するサブプログラムを図8.3に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(3-D1) アノテーションを付記したクラスファイル

クラスファイルのうち，メソッドのアトリビュート領域にアノテーション情報が付加されたクラスファイル．

(3-D2) AST

OpenJIT バイトコードディスコンパイラ機能で出力されたプログラムグラフ (AST)．

(3-D3) AST に対する付加データ

アノテーション情報から得られた，(2-D3) の AST に付加すべき情報．

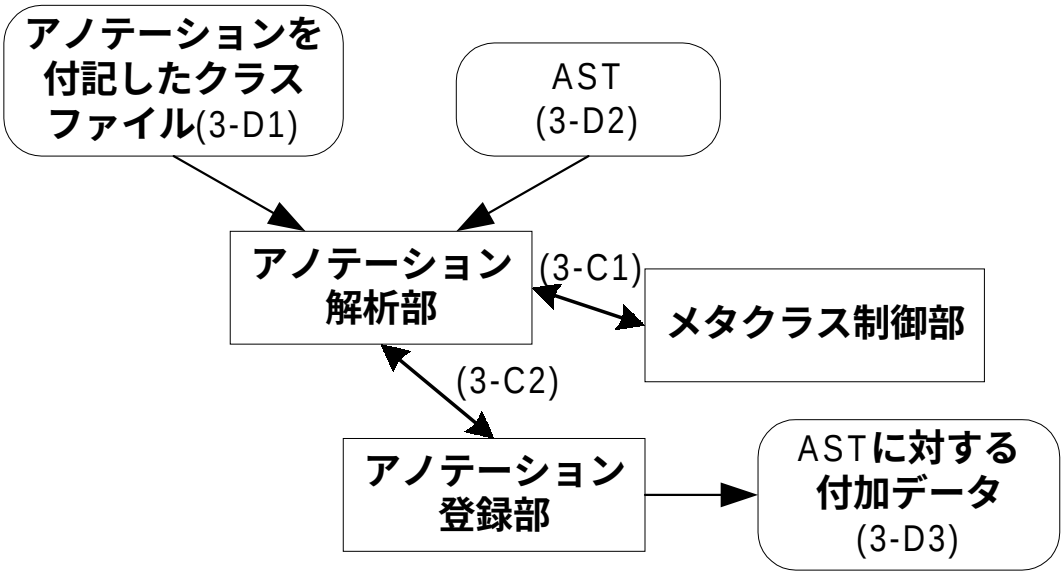

図 8.3: OpenJIT クラスファイルアノテーション解析機能

(4) OpenJIT 最適化機能

OpenJIT 最適化機能を構成するサブプログラムを図 8.4に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(4-D1) 最適化されたバイトコード

OpenJIT プログラム変換モジュールによって最適化された AST に対して，バイトコード出力部がバイトコードに変換を行う．その結果，得られた (7-D3) のバイトコード．

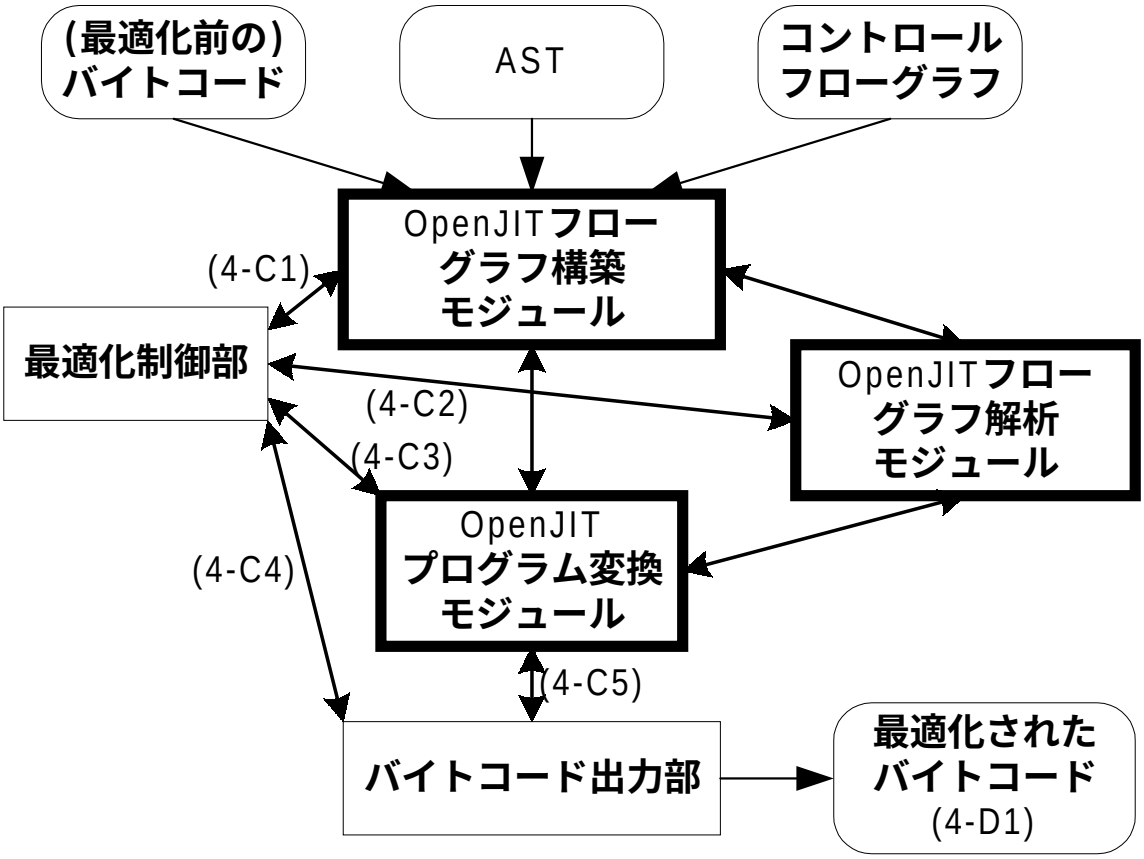

図 8.4: OpenJIT 最適化機能

(5) OpenJIT フローグラフ構築機能

OpenJIT フローグラフ構築機能を構成するサブプログラムを図8.5に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(5-D1) AST

OpenJIT バイトコードディスコンパイラ機能が出力した (2-D3) の AST ないし，OpenJIT クラスファイルアノテーション解析機能が出力した (3-D3) のデータ．

(5-D2) コントロールフローグラフ

OpenJIT バイトコードディスコンパイラが出力した，(2-D2) のコントロールフローグラフ．

(5-D3) クラスファイル間のクラス階層情報

a priori に与えられるクラスファイル間のクラス階層情報．

(5-D4) データフローグラフ

データフローグラフ構築部が出力するデータフローグラフ．

(5-D5) コントロール依存グラフ

AST 等入力部に与えられた AST，コントロールフローグラフから，コントロール依存グラフ構築部が生成したコントロール依存グラフ．

(5-D6) クラス階層グラフ

クラスファイル間のクラス階層情報と AST の情報から，クラス階層解析部が生成したクラス階層グラフ．

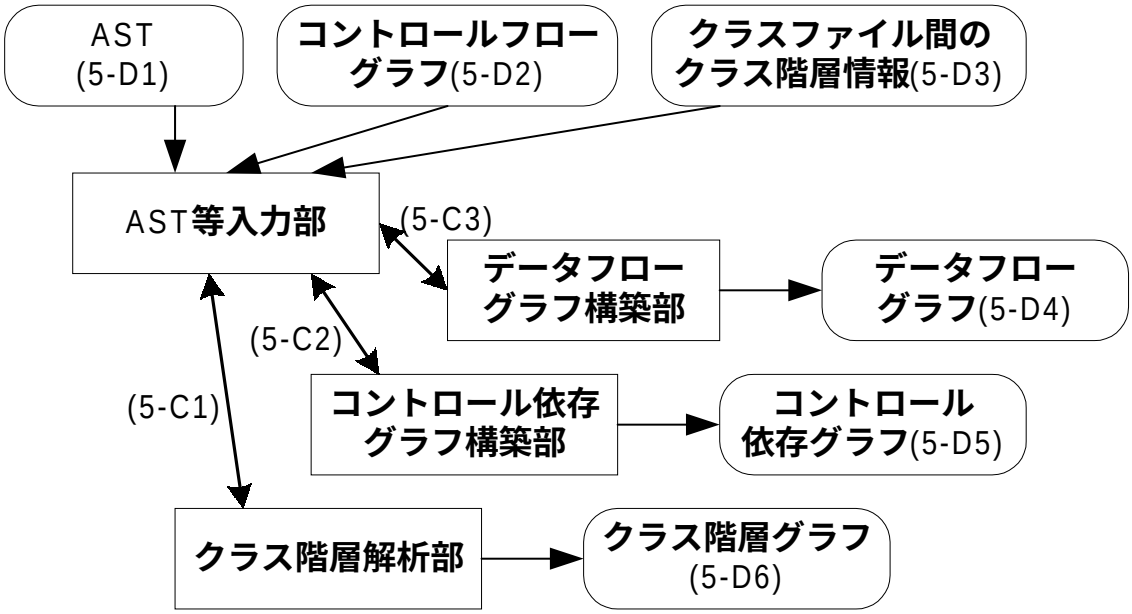

図 8.5: OpenJIT フローグラフ構築機能

(6) OpenJIT フローグラフ解析機能

OpenJIT フローグラフ解析機能を構成するサブプログラムを図8.6に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(6-D1) コントロールフローグラフ

OpenJIT バイトコードディスコンパイラが出力した，(2-D2) のコントロールフローグラフ．

(6-D2) コントロール依存グラフ

OpenJIT フローグラフ構築機能が出力した (5-D5) のコントロール依存グラフ．

(6-D3) データフローグラフ

OpenJIT フローグラフ構築機能が出力した (5-D4) のデータフローグラフ．

(6-D4) クラス階層グラフ

OpenJIT フローグラフ構築機能が出力した (5-D6) のクラス階層グラフ．

(6-D5) プログラム解析結果データ

フローグラフ上の解析を行った結果のデータ．

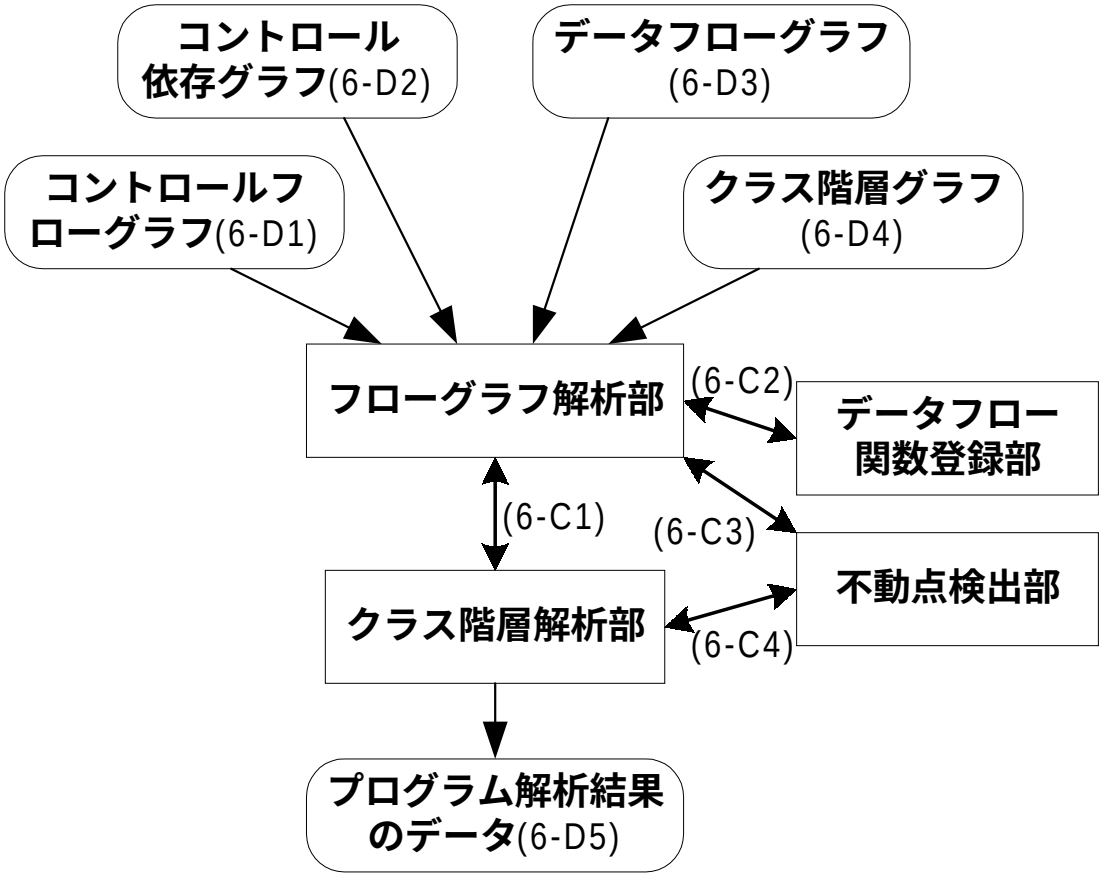

図 8.6: OpenJIT フローグラフ解析機能

(7) OpenJIT プログラム変換機能

OpenJIT プログラム変換機能を構成するサブプログラムを図 8.7に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(7-D1) AST

OpenJIT バイトコードディスコンパイラ機能が出力した (2-D3) の AST ないし，OpenJIT クラスファイルアノテーション解析機能が出力した (3-D3) のデータ．

(7-D2) プログラム解析結果のデータ

OpenJIT フローグラフ解析機能が出力した，フローグラフ上の解析を行った結果のデータ．

(7-D3) 変換された AST

元の AST から，パターンマッチによって変換された結果の AST．

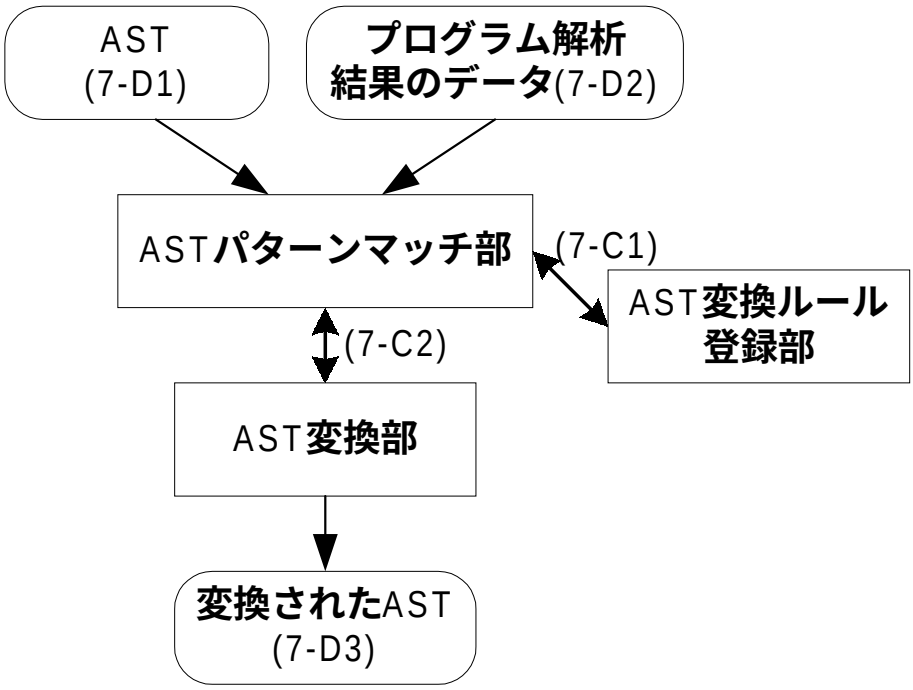

図 8.7: OpenJIT プログラム変換機能

8.2 OpenJIT バックエンドシステム

8.2.1 概要

OpenJIT バックエンドシステムにおける入出力データの仕様を以下に記述する.

8.2.2 入出力データ仕様

(1) OpenJIT ネイティブコード変換機能

OpenJIT ネイティブコード変換機能を構成するサブプログラムを図8.8に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(8-D1) バイトコード

OpenJIT フロントエンドシステムによる変換後のバイトコード．

(8-D2) SPARC ネイティブコード

SPARC プロセッサのネイティブコード．

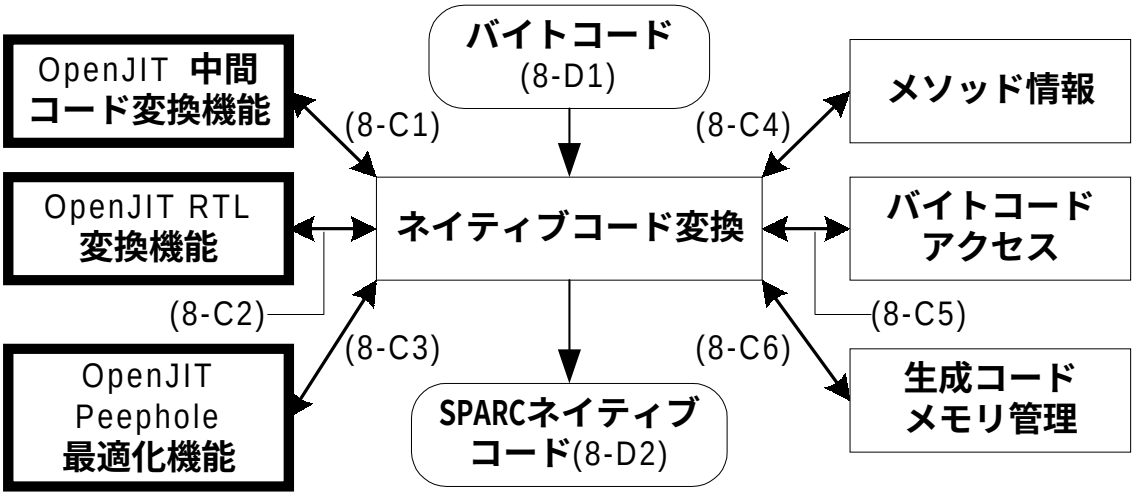

図 8.8: OpenJIT ネイティブコード変換機能

(2) OpenJIT 中間コード変換機能

OpenJIT 中間コード変換機能を構成するサブプログラムを図 8.9に示す．これらサブプログラム間で用いられる入出力データの一覧を示す.

(9-D1) バイトコード

OpenJIT フロントエンドシステムによる変換後のバイトコード.

(9-D2) 中間言語

バイトコードをパースして得られる中間言語.

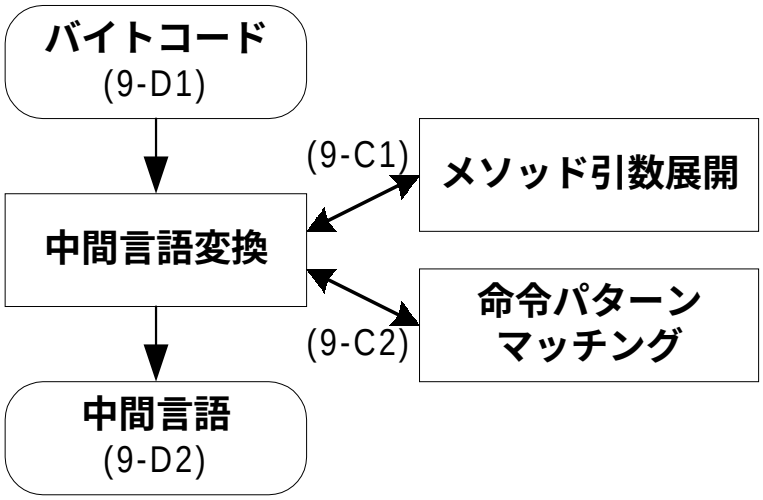

図 8.9: OpenJIT 中間コード変換機能

(3) OpenJIT RTL 変換機能

OpenJIT RTL 変換機能を構成するサブプログラムを図 8.10に示す．これらサブプログラム間で用いられる入出力データの一覧を示す.

(10-D1) 中間言語

OpenJIT 中間コード変換機能で出力された中間言語 (9-D2).

(10-D2) 基本ブロック情報

中間言語に対して，基本ブロック分割を行い，その結果として得られる基本ブロック情報.

(10-D3) RTL

基本ブロック情報を用いてコントロールフロー解析を行って，中間言語を変換することによって得られる RTL.

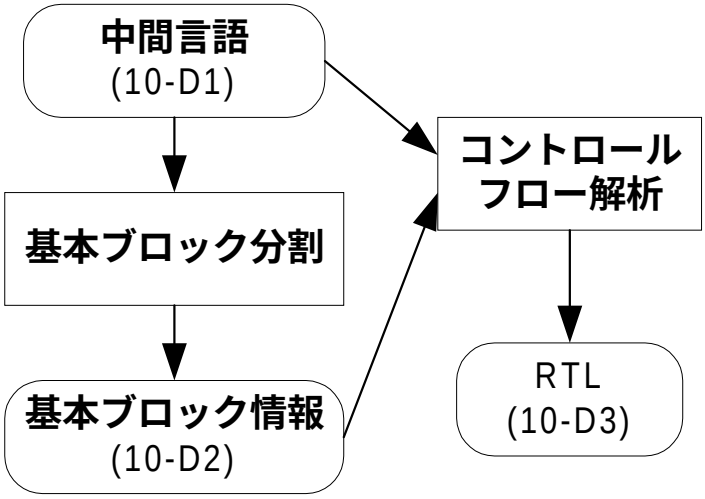

図 8.10: OpenJIT RTL 変換機能

(4) OpenJIT Peephole 最適化機能

OpenJIT Peephole 最適化機能を構成するサブプログラムを図 8.11に示す．これらサブプログラム間で用いられる入出力データの一覧を示す．

(11-D1) 基本ブロック情報

OpenJIT RTL 変換機能で生成された基本ブロック情報 (10-D2).

(11-D2) RTL

OpenJIT RTL 変換機能で生成された RTL(10-D3).

(11-D3) 最適化された RTL

各種 Peephole 最適化によって RTL 上の変換を行う．その結果得られる RTL.

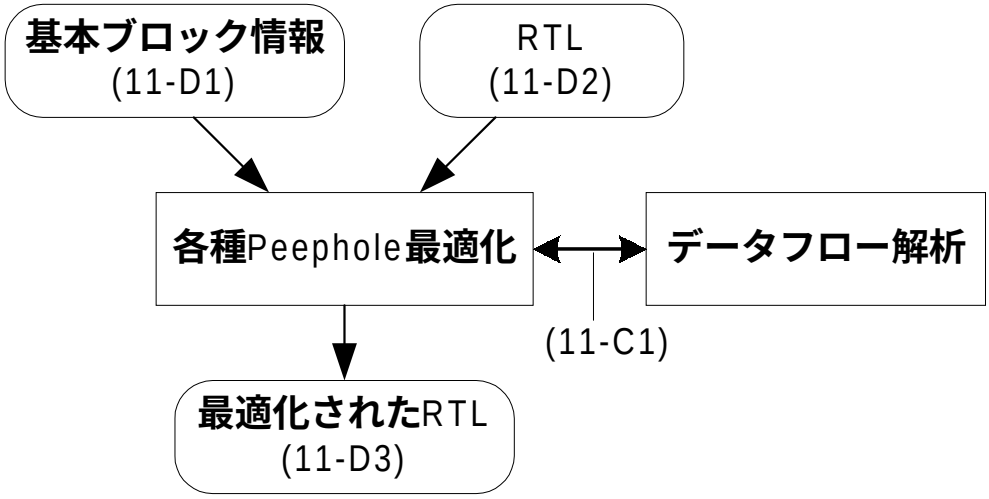

図 8.11: OpenJIT Peephole 最適化機能

(5) OpenJIT レジスタ割付機能

OpenJIT レジスタ割付機能を構成するサブプログラムを図 8.12に示す．これらサブプログラム間で用いられる入出力データの一覧を示す.

(12-D1) 仮想レジスタ番号

RTL で用いている仮想的なレジスタ番号.

(12-D2) スピル / フィルコード

レジスタ割り付けを行った結果，仮想レジスタ番号に該当する変数がレジスタに割り付けられない場合に出力される，スピル / フィルコード.

(12-D3) 物理レジスタ番号

レジスタ割り付けを行った結果，仮想レジスタ番号に該当する変数がレジスタに割り付けられる場合に出力される，レジスタ番号.

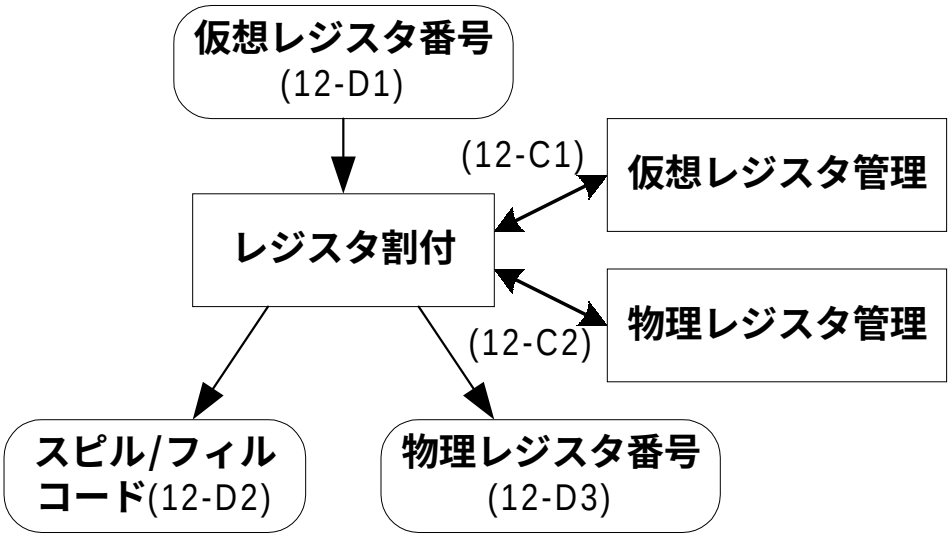

図 8.12: OpenJIT レジスタ割付機能

第 9 章

ファイル仕様

9.1 概要

ファイル仕様としては以下のものがある.

- Java Class ファイル

9.2 論理ファイル仕様

- Java Class ファイル
  The Java Virtual Machine Specification[1] 参照.

9.3　論理データベース仕様

本システムの場合は該当しない.

参考文献

[1] Sun Microsystems, Inc. “The Java Virtual Machine Specification”, 1996.