(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号

特許第4662657号
(P4662657)

(45)発行日　平成23年3月30日(2011.3.30)　(24)登録日　平成23年1月14日(2011.1.14)

(51)Int.Cl.			F I			
G06F	***9/44***	***(2006.01)***	G06F	9/44	530P
G06F	***9/45***	***(2006.01)***	G06F	9/44	322Z

請求項の数 6　(全 24 頁)

(21)出願番号	特願2001-209530(P2001-209530)
(22)出願日	平成13年7月10日(2001.7.10)
(65)公開番号	特開2002-73349(P2002-73349A)
(43)公開日	平成14年3月12日(2002.3.12)
審査請求日	平成18年12月27日(2006.12.27)
審判番号	不服2007-27392(P2007-27392/J1)
審判請求日	平成19年10月4日(2007.10.4)
(31)優先権主張番号	09/613289
(32)優先日	平成12年7月10日(2000.7.10)
(33)優先権主張国	米国(US)
(31)優先権主張番号	09/614158
(32)優先日	平成12年7月11日(2000.7.11)
(33)優先権主張国	米国(US)

早期審査対象出願

(73)特許権者　500046438
マイクロソフト　コーポレーション
アメリカ合衆国　ワシントン州　９８０５２－６３９９　レッドモンド　ワン　マイクロソフト　ウェイ
(74)代理人　100077481
弁理士　谷　義一
(74)代理人　100088915
弁理士　阿部　和夫
(74)復代理人　100115624
弁理士　濱中　淳宏
(74)復代理人　100115635
弁理士　窪田　郁大

最終頁に続く

(54)【発明の名称】統一データ型システムおよび方法

(57)【特許請求の範囲】

【請求項１】

異なるプログラム記述言語で記述された複数のソースコードファイルを、システム上に実装されたコンパイラを用いて出力コードに変換して実行するコンピューティングデバイスであって、

前記デバイスは、前記デバイスに接続されたコンピュータ読み取り可能な記憶媒体から、一のソースコードファイルを読み出し、

前記コンピューティングデバイスは、読み出したソースコードファイルを、前記コンパイラに受け渡し、

前記コンパイラは、前記ソースコードファイルを分析し、

前記ソースコードファイル中で定義されている値型を、前記コンピュータ読み取り可能な記憶媒体に記憶された、型ルールを用いて識別し、

前記ソースコードファイル中で定義されている値型がアンボックス化値型表現である場合、型ルールに含まれるメタデータを前記アンボックス化値型表現に追加してボックス化値型表現を生成し、前記値型がボックス化値型表現の場合、前記ボックス化値型表現からメタデータを取り除いて、アンボックス化値型表現を生成し、

前記ソースコードファイルを、前記アンボックス化値型表現と前記ボックス化値型表現の両方を含む出力コードに変換して出力し、

前記出力コードを実行することを特徴とするコンピューティングデバイス。

【請求項２】

異なるプログラム記述言語で記述された複数のソースコードファイルを、システム上に実装されたコンパイラを用いて出力コードに変換して実行するコンピューティングデバイスにおいて実行される方法であって、
前記デバイスが、前記デバイスに接続されたコンピュータ読み取り可能な記憶媒体から、一のソースコードファイルを読み出すステップと、
読み出したソースコードファイルを、前記コンピューティングデバイスから前記コンパイラに受け渡すステップと、
前記コンパイラが、前記ソースコードファイルを分析するステップと、
前記コンパイラが、前記ソースコードファイル中で定義されている値型を、前記コンピュータ読み取り可能な記憶媒体に記憶された、型ルールを用いて識別するステップと、
前記コンパイラが、前記ソースコードファイル中で定義されている値型がアンボックス化値型表現である場合、型ルールに含まれるメタデータを用いてボックス化値表現を生成し、前記値型がボックス化値型表現の場合、前記ボックス化値型表現からメタデータを取り除いてアンボックス化値型表現を生成するステップと、
前記コンパイラが、前記ソースコードファイルを、前記アンボックス化値型表現と前記ボックス化値型表現の両方を含む出力コードに変換して出力するステップと、
前記コンピューティングデバイスが、前記出力コードを実行するステップと
を含むことを特徴とする方法。

【請求項３】
請求項２に記載の方法であって、前記メタデータは前記ボックス化値型のインタフェースのリストを含むことを特徴とする方法。

【請求項４】
請求項３に記載の方法であって、前記コンピューティングデバイスは入力デバイスをさらに備え、前記メタデータは、前記入力デバイスからの信号によって定義されることを特徴とする方法。

【請求項５】
請求項４に記載の方法であって、前記コンピュータ読み取り可能な記憶媒体は、複数のオブジェクトクラスを含むオブジェクトクラス階層をさらに有し、前記ボックス化値型表現は、前記オブジェクトクラス階層の基本オブジェクトクラスから一つまたはそれ以上のインタフェースを継承することを特徴とする方法。

【請求項６】
請求項２に記載の方法であって、前記方法はランタイムに実行されることを特徴とする方法。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明は、データ型を定義し、処理するためのシステムおよび方法に関し、さらに具体的には、コンパイラおよび/またはランタイム（実行時）環境で使用される型システムに関する。

【０００２】

【従来の技術】

ほとんど当初から、コンピュータプログラミング言語には、データ型（data type）の考え方が具現化されていた。データ型には、文字、ストリング（文字列）、整数、浮動（float）などのような基本的概念が取り入れられている。その最も低いレベルでは、コンピュータにストアされたデータは、特定サイズのロケーション（例えば、32ビットのメモリロケーション）にストアされた単純ビットパターンになっている。データ型は、このビットパターンがどのように解釈されるかという考え方を定義したものである。例えば、特定サイズのストレージロケーション（記憶位置）に置かれた特定のビットパターンは、ストレージロケーションに文字が置かれているとした場合と、ストレージロケーションに「整数」が置かれているとした場合とでは、解釈の仕方が異なっている。

【０００３】
いくつかのコンピュータ言語では、データ型の考え方は存在していても、異なるデータ型をコンピュータプログラムの式の中に混在させるためにコンパイラまたはいずれかの関連ランタイムによって適用されるルールは、ほとんど存在していない。そのために、例えば、Cプログラミング言語では、整数値が浮動小数点数値で乗算されることがあっても、コンパイラエラーが引き起こされることがない。種々タイプのエラーを最小限にするために、この種の言語の多くでは、型ルールが組み込まれていて（built-in type rules)、ある種のデータ型を黙示的に変換することを可能にしている。他の場合には、あるデータ型を別のデータ型に「強制変換」または変換するための明示的構文（construct）が組み込まれている言語もある。言うまでもなく、この種の言語にはかなりの柔軟性があるが、種々のプログラミング式の中でデータ型を混在させるときに配慮を欠くと、ある種のプログラミングエラーが引き起こされることがある。

【０００４】
強く型付けされた言語（strongly typed language）では、厳格な型付けルール（strict typing rules）を提供することによって、プログラミングエラーの発生を低減化することが試みられている。強く型付けされた言語では、データ型の不一致が検出されると、コンパイラエラーが引き起こされるようになっている。例えば、Pascalでは、プログラマが文字値を整数変数に割り当てようすると、コンパイラエラーが引き起こされるようになっている。このようにすると、ある種のプログラミングエラーを低減化する効果があるが、ルールが余りに限定的になっている。

【０００５】
オブジェクト指向プログラミング言語の出現に伴い、データ型の考え方は別の意味をもつようになった。オブジェクト指向言語では、オブジェクトは、オブジェクトクラス階層（object class hierarchy）で表現されるのが代表的であり、そこでは、あるオブジェクトは、他の「基底クラス(base class)」のオブジェクトからのフィールド（プロパティとも呼ばれる）とメソッドから派生されている（またはこれらを継承している）。これらの言語におけるオブジェクトは、フィールド（特定データ型の変数で表されているのが代表的）と、これらのフィールドの操作を可能にする、あるいはある種の機能性をもつメソッド（method）または関数（function）とが混在することが可能になっている。さらに、オブジェクト指向言語には、浮動（float)、整数（integer)、文字（character)、ストリング（string）といったような、いくつかの組み込みデータ型（built-in data types）が用意されており、これらは、オブジェクトの中で基本変数としても、フィールドとしても使用できるのが代表的になっている。従って、例えば、Java(登録商標)では、プログラマは、整数型の変数を定義し、その１つが「整数」データ型になっているフィールドをもつオブジェクトを定義することが可能になっている。

【０００６】
オブジェクト指向プログラミング言語では、オブジェクトと基本データ型は、異なった扱い方をすることが可能になっている。例えば、単一プロパティが整数型であるオブジェクトと整数型の変数は、実際には、どちらも整数を表しているのにすぎないのに、多くのオブジェクト指向言語では、同じデータ型でないものと扱われることになる。整数型の変数は、特定ストレージロケーションにビットパターンとして存在しているのにすぎず、追加情報が置かれていないのに対し、オブジェクトは、同じサイズのストレージロケーションと、そのストレージロケーションに置かれた値がどのように解釈されるかを記述している追加情報（つまり、メタデータ(metadata)）をもっている。

【０００７】
オブジェクト表現と基本データ型表現の間にある種の等価性をもたせるために、「ボックス化(boxing)」という考え方が考案された。基本データ型表現にメタデータを追加してオブジェクト表現を得るためのプロセスは、「ボックス化」と呼ばれている。同様に、オブジェクト表現からメタデータを取り除いて基本データ型表現を得るためのプロセスは「アンボックス化(unboxing)」と呼ばれている。しかし、ボックス化とアンボックス化が開発

されたとしても、現存のコンパイラおよび/またはランタイムシステムはデータ型を分割した考え方を採用し、オブジェクトの考え方と基本データ型表現の考え方が厳格に区別されている。この区別には多くの意味合いがあるが、その意味合いが叙述に現れている１つの分野は、これらの言語がユーザ定義型（user-defined types）をどのように扱うか、という点にある。

【０００８】

オブジェクト指向プログラミング以前であっても、大部分ではないが、多くのプログラミング言語は、ユーザ定義データ型の考え方を取り入れていた。これらのプログラミング言語によれば、プログラマは、その言語の基本組み込み型から新しい「データ型」を作ることを可能にしていた。例えば、プログラマは、新しい型である"data_point"（データポイント）を、浮動型のx座標値と浮動型のy座標値からなるものとして定義することが可能であった。しかし、Java(登録商標)のような、ある種のオブジェクト指向プログラミング言語では、基本組み込み型を上記のように拡張することを許していない。この種の、いくつかの実装（implementation）では、ユーザ定義型が許されているのは、オブジェクトの形体にあるときだけである。ランタイム時に適用可能な統一データ型システム（unified data type system）が要望されているにもかかわらず、既存の解決方法はこの要望に十分に応えていない。

【０００９】

【発明が解決しようとする課題】

本発明は、なかんずく、データ型を分割するという、現在の見方を避けるメカニズムを取り扱うことを課題にしている。また、本発明は、オブジェクト型の方が効率的であるような場合に基本データ型を使用し、基本データ型の方が効率的であるような場合にオブジェクト型を使用すると起こる非効率性を解決することも課題にしている。

【００１０】

【課題を解決するための手段】

本発明によれば、上記およびその他の課題は、ユーザ定義のデータ型を効率的に処理するシステムおよび方法を提供することによって解決されている。本発明によれば、プログラミング言語、特にオブジェクト指向プログラミング言語の型システムがより統一化された見方で提供されている。本発明の型システムでは、基本データ型には２つの表現（二重表現(dual representation)）が用意されている。一方の表現は、基本組み込みデータ型に共通する基本データ型表現である。本明細書では、この表現は、値型（value type）表現、もっと簡単には、値型と呼ぶことにする。しかし、他の型システムと異なり、基本データ型の各々には、型システム自身のオブジェクト階層に存在するボックス化表現（boxed representation）も用意されている。この二重表現は、ユーザ定義型（user-defined types）にも拡張できるので、ユーザ定義型は、型システムのオブジェクト階層内に値型としても、オブジェクトとしても存在することが可能になっている。そのため、コンパイラおよび/またはランタイムは、コンパイル（ランタイム）時の特定の要求に応じて、最も効果的で、最も効率的なデータ型の表現を選択することが可能になっている。

【００１１】

データ型の二重表現のほかに、本発明の別の形態によれば、あるデータ型のボックス化表現をいつ使用するか、そのデータ型の値型（またはアンボックス化）表現をいつ使用するかを判断するためのルールの適用が可能になっている。これらのルールは、例えば、コンパイラによって適用できるので、なかんずく、ある特定データ型のボックス化表現とアンボックス化表現の間で黙示的に変換することが可能になっている。

【００１２】

本発明の別の形態によれば、統一化された型システムの見方は、オブジェクトの仮想メソッド（virtual method）の振る舞い（behavior　作用ともいう）に反映されている。オブジェクトの基本的特徴の１つは、「親」オブジェクトからメソッドを継承できることである。そのようなメソッドとして、オブジェクトを引数（argument）として受け取るメソッドがある。値型が、階層内に値型としても、オブジェクトとしても表現できること（二重

表現）は、値型がメソッドをもつことができ、ある場合にはオブジェクトとして、他の場合には値型として振る舞うことができることを意味している。詳細は、以下でもっと詳しく議論するが、その実用的効果は、値型がボックス化表現になっているとき、他のオブジェクトと同じように型情報をもつことができることである。さらに、値型がアンボックス化表現になっているとき、メソッドがオブジェクト型（ボックス化表現など）を予想しているときでも、値型がそのメソッドに対して有効な引数となることができることである。このアプローチによると、完全に新規で、強力なプログラミングパラダイムが開発者に提供されることになる。さらに、ボックス化表現とアンボックス化表現の両方が用意されているので、どちらのバージョンの値型（つまり、ボックス化表現かアンボックス化表現か）を使用するか、あるいは一方の型から他方の型に変換することを、開発者がソースコードの中で明示的に指定しなくても、この機能のすべてが利用可能になっている。

【００１３】

本発明の一実現形態では、統一型システムは、ランタイム環境で提供されている。ソースコードファイルには、アンボックス化値型表現が含まれている。アンボックス化値型表現には、アンボックス化値表現をボックス化値型表現に変換するためのメタデータが関連付けられている。出力コードは、ランタイムオペレーション時に異なる型が検出されると、コンパイラがアンボックス化値型表現とボックス化値型表現の間で変換を行うことにより生成される。

【００１４】

本発明の別の実現形態では、少なくとも１つのアンボックス化値型表現を含んでいるソースファイルを変換するための方法が提供されている。この方法によれば、ソースファイルがアンボックス化値型表現を含んでいるかが判断される。この判断オペレーションに応じて、メタデータがアンボックス化値型表現に関連付けられている。異なる型をもつオペランドのあるオペレーションは、ソースファイルの中に指定されている。一方のオペランドはアンボックス化値型表現に、他方のオペランドはボックス化値型表現になっている。出力コードは、オペランドの一方が他方のオペランドの型と一致するようにコンパイラが変換することによって得られる。

【００１５】

本発明のその他の実現形態では、製造物品がコンピュータプログラムプロダクトとして提供されている。コンピュータプログラムプロダクトの一実施形態では、コンピュータシステムによって読み取り可能であるコンピュータプログラム記憶媒体が提供され、そこには、少なくとも１つのアンボックス化値型表現を収めているソースファイルをコンパイルするためのコンピュータプログラムがコーディングされている。コンピュータプログラムプロダクトの別の実施形態はコンピュータデータシグナルの形で提供され、このシグナルは、コンピューティングシステムによって搬送波で具現化され、少なくとも１つのアンボックス化値型表現を収めているソースファイルをコンパイルするためのコンピュータプログラムがコーディングされている。コンピュータプログラムプロダクトは、少なくとも１つのアンボックス化値型表現を収めているソースファイルをコンパイルするためのコンピュータプロセスを、コンピュータシステム上で実行するためのコンピュータプログラムをコーディングしている。ソースファイルがアンボックス化値型表現を含んでいるかどうかが判断される。この判断オペレーションに応じて、メタデータがアンボックス化値型表現に関連付けられている。異なる型をもつオペランドのあるオペレーションは、ソースファイルの中に指定されている。一方のオペランドはアンボックス化値型表現に、他方のオペランドはボックス化値型表現になっている。出力コードは、オペランドの一方が他方のオペランドの型と一致するようにコンパイラが変換することによって得られる。

【００１６】

本発明のさらに別の形態では、上記考え方はランタイムまたは実行環境と結合され、値型、オブジェクトクラス、およびインタフェースをサポートするユニークなランタイム環境が得られるようにしている。

【００１７】

本発明を特徴付けている、上記およびその他の種々特徴と利点を分かりやすくするために、添付図面を参照して以下で詳しく説明する。

【００１８】

【発明の実施の形態】

本発明の例示実施形態によれば、プログラミング言語、特にオブジェクト指向プログラミング言語の型システムが、より統一化された見方で提供されている。例示の型システムでは、基本データ型には２つの表現（二重表現）が用意されている。一方の表現は、アンボックス化値型、単純には値型と呼ばれる基本データ型表現である。アンボックス化値型には、コンパイラから出力される出力コードの中で型情報が付いていないのが一般的である。しかるに、本発明の実施形態では、基本データ型の各々はボックス化表現ももち、これは型システムのオブジェクト階層に置かれ、コンパイラから出力された出力コードの中で型情報が付けられている（例えば、メタデータで指定されている）。この二重表現はユーザ定義型にも拡張可能であるので、ユーザ定義型は、型システムのオブジェクト階層内にアンボックス化値型としても、オブジェクト（つまり、ボックス化値型）としても存在することが可能になっている。そのため、コンパイラおよび/またはランタイムは、コンパイル（ランタイム）時の特定要求に応じて、最も効果的で、最も効率的な値型表現を選択することが可能になっている。

【００１９】

図１は、本発明の実施形態において型システムを統一化された見方で提供するように動作するコンパイラの論理的表現を示す図である。ソースファイル１００は、ある特定のプログラミング言語規格に準拠するように作成されたソースプログラムを表しており、この中には、C言語やC++言語、その他の高水準プログラム言語または中間言語に関する規格も含まれるが、これらに限定されないことは勿論である。ソースファイル１００は、ランタイム環境で１つまたは２つ以上のオペレーション（演算、操作など）を実行するための命令とデータを含むことも可能である。図示の実施形態では、ソースファイル１００はコンパイラ１０４によって受け取られ、そこでソースコードは出力コード（例えば、オブジェクトコードまたは実行可能コード）に変換される。本発明の代替実施形態では、当然に理解されるように、コンパイラ１０４は、図６に示すようにライタイム環境１０２（例えば、Just-In-Time (JIT) コンパイラとして）に組み込んでおくことも可能である。

【００２０】

本発明の代替実施形態では、ソースファイル１００のソースコードは、中間言語コードファイル１０６で示すように、中間言語に変換されてからコンパイラ１０４に渡されるようにすることも可能になっている。以下の説明で扱っている実施形態では、ソースファイル１００がコンパイラ１０４に入力されるようになっているが、当然に理解されるように、本発明の実施形態では、ソースコードでも、中間コードでもコンパイラ１０４に入力できることを目的としている。同様に、ソースコードと中間コードは、データおよび関連データ型を定義するための構造と構文に互換性をもたせることが可能であり、これは本発明の範囲に属するものである。

【００２１】

一般的に、コンパイラは、ソースコード（または中間言語コード）をオブジェクトコードまたは実行可能コードに変換するプログラムである。コンパイラという名前の由来はその働き方にある。すなわち、コンパイラはソースコードの断片全体を調べていき、そこに含まれる命令とデータを収集し、再編成する。ある実装では、第２のステージには、コンパイルされたオブジェクトコードを他のオブジェクトコードとリンクして、実行可能プログラムを生成するリンカが含まれている。他の実装では、このリンキングプログラムは、ランタイム直前またはランタイム時に実行されるので、これは「実行時バインディング (late binding)」とも、「ランタイムバインディング (runtime binding)」とも呼ばれている。

【００２２】

上述したように、プログラミング言語は、データ型の考え方を採用しているのが代表的で

ある。ソースコード１００内のデータは、一般的に、２つのデータ型からなっている。(1) 値型 (value type) １１０と (2) オブジェクト１１２である。理解しやすくするために、以下に説明するクラスとオブジェクト名は、大文字の名前で示され、値型と値型変数名は小文字の名前が付けられている。データは、変数名および関連の型インジケータを使用して「値型」としてソースファイルに定義することが可能である。例えば、インデックス（指標）を表すデータは、"int index" として定義することができる。ここで、"int" はデータ型インジケータ、"index" は変数名である。別の方法として、データは、オブジェクト名、クラスインジケータ、およびクラス定義を使用して「オブジェクト」としてソースファイルに定義することも可能である。例えば、下に記述されている例示のソースコードは、Rectと名付けたクラスを定義しており、そこには、矩形の４隅を定義している４つのデカルト座標 (Cartesian coordinate) が含まれている。

【００２３】

【数１】

```
class Rect
{
      Cartesian UpperLeft;
      Cartesian UpperRight;
      Cartesian LowerLeft;
      Cartesian LowerRight;
      float       area;
};

Rect      RectObject;
```

【００２４】

ステートメント"Rect RectObject"は、クラス"Rect"（クラスインジケータ）のオブジェクト"RectObject"（オブジェクト名）を定義している。なお、RectObjectは、オブジェクト"UpperLeft"、"UpperRight"、"LowerLeft"、および"LowerRight"のように、クラスCartesian（これのクラス定義は示されていない）の他のオブジェクトと値型、およびデータ型"float"の値型"area"を含むことも可能である。また、当然に理解されるように、上記説明は、ソースファイル内のデータの定義例を示したものにすぎず、データ定義の他のデータ構造と構文も本発明の範囲に含まれるものである。

【００２５】

組み込みデータ型１１４と基本オブジェクトクラス階層を含む、種々の組み込みデータ型の１つまたは２つ以上、および種々データ型相互間で変換し、データ型相互間の関係を定義するための型ルール (type rules) １１８を、コンパイラ１０４に組み入れておくこともできるが、コンパイラにこれらにアクセスさせるようにすることも可能である。一般的に、組み込み値型とは、整数を表す"int"、文字を表す"char"、浮動小数点数を表す"float"のように、プログラミング言語に基本的なものであり、プログラムによって共通に使用されると考えられている値型のことである。

【００２６】

同様に、基本オブジェクトクラス階層１１６には、基本的で、共通に使用されるクラスが継承階層に用意されている。例えば、階層内のルート (root) クラスは、"BasicObject" を定義することが可能であり、そこには、プログラミング言語における基本オブジェクトの基本的特性（例えば、データと関数）が含まれている。ルートクラスの子供は、その用途をさらに具体化するためにBasicObjectクラスを「継承」または「拡張」するように定

義することができる。例えば、"Shape"クラスと"Point"クラスはBasicObjectクラスから継承することができ、"Rect"クラスと"Circle"クラスは"Shape"クラスから継承することができる。基本クラスを結合したものが、オブジェクトクラス階層を構成している。オブジェクトクラス階層の別の例は図４に示されているが、これについては以下で説明する。

【００２７】

多くのプログラミング言語では、組み込み値型と基本クラスは共に拡張可能またはカストマイズ可能になっている。例えば、C言語では、開発者はキーワード"typedef"を使用して、新しい値型を定義することができる。例えば、値型"coordinate"は、以下に示すように、デカルト平面（Cartesian plane）上のX-Y座標を表している、２つの浮動小数点数を含んでいる構造として定義することができる。

【００２８】

【数２】

```
typedef struct
{
  float   x;
  float   y;
} coordinate;
```

【００２９】

同様に、ソースコードは、基本クラスの１つまたは２つ以上を継承または拡張することによって基本オブジェクトクラス階層を拡張することができる。例えば、ユーザ定義のオブジェクトは、"CustomShape"クラスを定義するように基本Shapeクラスを拡張することができる。図１に戻って説明すると、組み込み値型とユーザ定義の値型は共に、値型１１０によってソースコードに表すことが可能であり、基本オブジェクトとユーザ定義のオブジェクトは共に、オブジェクト１１２によってソースコードに表すことが可能である。

【００３０】

本発明の実施形態では、コンパイラ１０４は、異なる値型間の変換を正しく行うための命令をコンパイラに与える型ルール１１８を備えることが可能である。例えば、C言語では、ソースコード命令は整数値を浮動小数点変数に割り当てることが可能になっている（例えば、"float amount = total;"。ここで、"total"は整数型のデータ値であり、"amount"は浮動小数点数変数である）。Cコンパイラは、型ルール１１８を適用して、整数"total"を浮動小数点数値に自動変換してから値を変数"amount"に割り当てる命令を生成するようにしている。これとは別に、値型とオペレーションの組み合わせが与えられているとき、該当の型ルールがコンパイラ１０４に用意されていない場合もある（例えば、"integer"値を"coordinate"変数に割り当てる）。このような場合には、コンパイラはコンパイラエラーを出すか、あるいはランタイム例外を引き起こしてエラーを通知することになる。

【００３１】

本発明の実施形態では、オペランドの一方または両方をボックス化および/またはアンボックス化することによって値型とオブジェクトの間でオペレーションを実行するためのソースコード命令は、開発者から見えないようにコンパイルされ、実行されることが可能になっている。第１の例では、ソースコード命令は、値型をもつ値が、オブジェクトに割り当てられることを示している（例えば、"integer"値型をもつ値は"Integer"クラスのオブジェクトに割り当てられる）。ボックス化なしでこの割り当てを行うと、オペランドの型が等価でないのでコンパイラエラーになるのが代表的である（つまり、この割り当ては型ルールに従っていない）。ボックス化/アンボックス化していないと、コンパイラが割り当てを行わないのは、"integer"値型が、Integerオブジェクトのメタデータ部分に移植（populate）するために必要なメタデータと関連付けられていないからである（図５を参照）。

【００３２】
その代わりに、コンパイラ１０４は型間に不一致があることを検出し、メタデータと、"integer"値を「ボックス化」するためのコンパイラコードを生成することにより、値型をオブジェクトに変換し、ボックス化値型がIntegerオブジェクトに割り当てられるようにする。型に組み込み値型が含まれている場合は、その値型をボックス化するために必要なメタデータでコンパイラを構成し直すことが可能である。型にユーザ定義の値型が含まれている場合は、ユーザは必要とされるメタデータを用意し、ボックス化オペレーションでコンパイラに使用させることができる。ユーザ定義の型を定義するメタデータは、ビット列（つまり、値）を記述することができ、その中には、型名、型に含まれるすべてのフィールドのフィールド名、すべてのフィールドのフィールドタイプ、およびその型に関連して実行できるオペレーション（例えば、メソッド）が含まれている。ユーザ定義の型を定義するメタデータには、その型がボックス化形式で実装しているインタフェースのリストを含めることも可能である。最後に、メタデータには、値型のボックス化バージョンが、オブジェクトクラス階層１１６内のどこに収まるかを示す標識を含めることができる。なお、これについては、図４を参照して以下で説明する。そのあと、コンパイラ１０４は、「ボックス化値型」（またはオブジェクト）をIntegerオブジェクトに割り当てるためのオブジェクトコードを生成する。
【００３３】
値型をボックス化し、アンボックス化するコードをコンパイラで生成するのではなく、他の実施形態によれば、二重表現の考え方をいくつかの異なる方法で実現することも可能である。例えば、コードのターゲットであるランタイム環境（例えば、ランタイム環境１０２など）が値型をボックス化し、アンボックス化できる場合には、コンパイラは、ボックス化またはアンボックス化コマンドのうち該当する方を出力するだけで済むので、ランタイムは実際の作業を行うことができる。他の実現形態では、ボックス化表現とアンボックス化表現は同時に存在できるので、値型をボックス化またはアンボックス化するコードをコンパイラにも、ランタイム環境にも生成させる必要がなくなる。また、他の実現形態では、アンボックス化バージョンが望ましいときメタデータ部分をバイパスまたは無視するメカニズムを使用することで、ボックス化表現だけを生成させることができる。
【００３４】
コンパイラ１０４によって生成される出力コード１０８は、コンパイルされたオブジェクト１２０と、ソースファイル１００の中で定義された値型のボックス化表現とアンボックス化表現（１２２と１２４）の両方を論理的に含んでいる。ある実施形態では、すべての値型は、ボックス化表現とアンボックス化表現の両方を論理的に生成するようにコンパイルされる。値型の二重表現自体は、事実上同じストレージロケーションに置いておくことができるが、代替実施形態では、別々のストレージロケーションに置いておくことも可能になっている。別の実施形態では、オブジェクトと作用し合う値型だけがボックス化されるので、出力コードのサイズが低減されることになる（つまり、不要なメタデータが省かれることになる）。さらに、本発明の別の実施形態では、一度に値型の１つのバージョン（つまり、ボックス化またはアンボックス化）を残しておくことができるので、２バージョン間の変換を必要時に動的に行うことができる。従って、値型の１つのバージョンだけを、いつでも残しておけばよいので、値型バージョンは、ある特定オペレーション（例えば、割り当て、オブジェクトパラメータを使用した関数コールなど）に従ってコンパイラによって生成されたコードで動的に変換されることになる。
【００３５】
さらに、出力コード１０８は、値型と関連付けられたメタデータを含むことが可能である。別の方法として、ボックス化オペレーションで作成されたオブジェクトのマシンまたは実行可能コード表現を出力１０８に含めておくことも可能である。さらに別の代替方法では、マシンまたは実行可能コード表現は、以下で説明するように、ランタイム時に生成することも可能である。
【００３６】

ボックス化および/またはアンボックス化によって値型とオブジェクト間のオペレーションを実行するソースコード命令の別の例に示すケースでは、ボックス化値型が、アンボックス化値型をもつ値に割り当てられるようになっている（例えば、Integerクラスのオブジェクトはinteger値型をもつ値に割り当てられる）。この種の割り当てをアンボックス化なしで行うと、オペランドの型が等価でないのでコンパイラエラーになる。しかるに、本発明の実施形態では、コンパイラ１０４は型間が不一致であることを検出し、ボックス化値型と関連付けられたメタデータを削除または無視することによってオブジェクトをアンボックス化値型に変換するコードを生成して、アンボックス化値型が"integer"値型に割り当てられるようにする。

【００３７】

図２と以下の説明は、本発明を実現するのに適しているコンピューティング環境の概要を要約して説明することを目的としている。本発明は、パーソナルコンピュータと併用してオペレーティングシステム上で実行されるアプリケーションプログラムを中心に説明されているが、この分野の精通者ならば当然に理解されるように、本発明は他のプログラムモジュールと組み合わせて実現することも可能である。一般的に、プログラムモジュールには、特定のタスクを実行する、または特定の抽象データ型を実装しているルーチン、プログラム、コンポーネント、データ構造などが含まれている。さらに、この分野の精通者ならば理解されるように、本発明は、他のコンピュータシステム構成で実施することも可能であり、その中には、ハンドヘルドデバイス、マイクロプロセッサシステム、マイクロプロセッサベースのコンシューマエレクトロニックスやプログラマブルコンシューマエレクトロニックス、ミニコンピュータ、メインフレームコンピュータなどが含まれている。本発明は、コミュニケーションネットワークを通してリンクされたリモートの処理デバイスによってタスクが実行されるような、分散コンピューティング環境で実施することも可能である。分散コンピューティング環境では、プログラムモジュールはローカルとリモートの両方のメモリストレージデバイスに置いておくことが可能である。

【００３８】

図２を参照して説明すると、本発明を実現するための例示のシステムは従来のパーソナルコンピュータ２０を含み、パーソナルコンピュータには、処理ユニット２１、システムメモリ２２、およびシステムメモリを処理ユニット２１に結合するシステムバス２３が搭載されている。システムメモリ２２には、リードオンリメモリ（read only memory (ROM)）２４とランダムアクセスメモリ（random access memory (RAM)）２５が含まれている。スタートアップ時のように、パーソナルコンピュータ２０内のエレメント間で情報を転送するのを支援する基本ルーチンを収めている基本入出力システム（basic input/output system (BIOS)）２６は、ROM２４に格納されている。パーソナルコンピュータ２０は、さらに、ハードディスクドライブ２７、例えば、取り外し可能ディスク２９との間で読み書きを行う磁気ディスクドライブ２８、および例えば、CD-ROMディスク３１を読み取ったり、他の光媒体との間で読み書きを行ったりする光ディスクドライブ３０を搭載している。ハードディスクドライブ２７、磁気ディスクドライブ２８、および光ディスクドライブ３０は、それぞれハードディスクドライブインタフェース３２、磁気ディスクドライブインタフェース３３、および光ディスクドライブインタフェース３４を通してシステムバス２３に接続されている。これらのドライブとそれぞれに関連するコンピュータ読み取り可能媒体は、不揮発性ストレージとしてパーソナルコンピュータ２０に利用されている。上記のコンピュータ読み取り可能媒体の説明では、ハードディスク、取り外し可能磁気ディスクおよびCD-ROMディスクが挙げられているが、この分野の精通者ならば理解されるように、磁気カセット、フラッシュメモリカード、デジタルビデオディスク、ベルヌーイ（Bernoulli）カートリッジなどのように、コンピュータによって読み取り可能である他のタイプの媒体を、例示の動作環境で使用することも可能である。

【００３９】

複数のプログラムモジュールをドライブとRAM２５に格納しておくことができる。その中には、オペレーティングシステム３５、ソースファイル１００、ランタイムシステム１０

２、およびコンパイラ１０４が含まれる。ユーザは、キーボード４０およびマウス４２などのポインティングデバイスを通して、コマンドと情報をパーソナルコンピュータ２０に入力することができる。その他の入力デバイス（図示せず）としては、マイクロホン、ジョイスティック、ゲームパッド、サテライトディッシュ、スキャナなどがある。上記および他の入力デバイスは、システムバスに結合されたシリアルポートインタフェース４６を通して処理ユニット２１に接続されていることが多いが、ゲームポートやユニバーサルシリアルバス（universal serial bus (USB)）などの、他のインタフェースを通して接続することも可能である。モニタ４７や他のタイプのディスプレイデバイスも、ビデオアダプタ４８などのインタフェースを通してシステムバス２３に接続されている。モニタのほかに、パーソナルコンピュータは、スピーカやプリンタのような、他の周辺出力デバイス（図示せず）を装備しているのが代表的である。

【００４０】

パーソナルコンピュータ２０は、リモートコンピュータ４９などの、１つまたは２つ以上のリモートコンピュータとの論理的コネクションを使用して、ネットワーキング環境で動作することができる。リモートコンピュータ４９は、サーバ、ルータ、ピアデバイスまたは他の共通ネットワークノードである場合があり、図２にはメモリストレージデバイス５０だけが示されているが、パーソナルコンピュータ２０に関連して上述したエレメントの多くまたは全部を搭載しているのが代表的である。図２に示す論理的コネクションには、ローカルエリアネットワーク（local area network (LAN)）５１と広域ネットワーク（wide area network (WAN)）５２が含まれている。この種のネットワーキング環境は、オフィス、企業内（enterprise-wide）コンピュータネットワーク、イントラネットおよびインターネット（the Internet）では日常的になっている。

【００４１】

LANネットワーキング環境で使用されるときは、パーソナルコンピュータ２０は、ネットワークインタフェース５３を通してLAN５１に接続されている。WANネットワーキング環境で使用されるときは、パーソナルコンピュータ２０は、インターネットのような、WAN５２上のコミュニケーションを確立するためのモデムや他の手段を搭載しているのが代表的である。モデム５４は内蔵されている場合と、外付けの場合があるが、シリアルポートインタフェース４６を介してシステムバス２３に接続されている。ネットワーキング環境では、パーソナルコンピュータ２０に関連して説明したプログラムモジュールまたはその一部は、リモートメモリストレージデバイスに格納しておくことができる。当然に理解されるように、図示のネットワークコネクションは例示であり、コンピュータ間の通信リンクを確立する他の手段を使用することも可能である。

【００４２】

パーソナルコンピュータ２０のようなコンピューティングデバイスは、少なくともなんらかの形のコンピュータ読み取り可能媒体を具備しているのが代表的である。コンピュータ読み取り可能媒体としては、パーソナルコンピュータ２０がアクセスできる、利用可能な媒体ならば、どの媒体でも可能である。一例として、コンピュータ読み取り可能媒体としては、コンピュータ記憶媒体と通信媒体があるが、これに限定されるものではない。コンピュータ記憶媒体には、コンピュータ読み取り可能命令やデータ構造、プログラムモジュール、その他のデータなどの情報を格納しておくために、なんらかの方法またはテクノロジで実現されている揮発性、不揮発性、取り外し可能および取り外し不能（固定）媒体がある。コンピュータ記憶媒体としては、RAM、ROM、EEPROM、フラッシュメモリや他のメモリテクノロジ、CD-ROM、デジタルバーサタイルディスク（digital versatile disk (DVD)）や他の光ストレージ、磁気カセット、磁気テープ、磁気ディスクストレージや他の磁気ストレージデバイス、あるいは必要とする情報を格納しておくために利用でき、パーソナルコンピュータ２０がアクセスできる他の媒体があるが、これらに限定されるものではない。通信媒体は、コンピュータ読み取り可能命令、データ構造、プログラムモジュールまたは他のデータを、搬送波や他のトランスポートメカニズムなどの変調データ信号で具現化しているのが代表的であり、その中には、すべての情報伝達媒体が含まれている。ここ

で、「変調データ信号(modulated data signal)」という用語は、信号に含まれる情報をエンコード（符号化）するような形で設定または変更された特性の１つまたは２つ以上をもつ信号を意味している。一例を挙げると、通信媒体には、有線ネットワーク (wired network) や直接有線コネクション (direct-wired connection) などの有線媒体 (wired media)、および音響やRF、赤外線、その他のワイヤレス（無線）媒体などのワイヤレス（無線）媒体があるが、これらに限定されない。上記に挙げたものを任意に組み合わせたものも、当然にコンピュータ読み取り可能媒体の範囲に含まれている。コンピュータ読み取り可能媒体は、コンピュータプログラムプロダクトと呼ばれることもある。

【００４３】

図１に関連して上述したように、コンパイラ１０４は、ランタイム環境１０２または他の任意の実行環境用に書かれたソースファイル１００を受け取り、コンパイルする。図３は、その言語でソースファイル１００が書かれているコンピュータ言語に対する値型分類システム (value type classification system) ３００の例を示す図である。ソースファイル１００は、組み込み値型３０２とユーザ定義の値型３０４の両方を利用することができる。一般的に、値型は、コンピュータにストアされたデータのビットパターンがどのように解釈されるかという考え方を定義したものである。例えば、値は、整数または浮動小数点数を表した単純なビットパターンである場合がある。各値は、値が占有しているストレージのサイズだけでなく、値表現内のビットの意味も記述している型をもっている。例えば、"2"の値は"int16"という型にすることができる。型"int16"は、値が整数であることを値表現のビットが意味していることを示している。さらに、型"int16"は、値が符号付き16ビット整数をストアするために必要なストレージを占有することを示している。また、型は、値表現に対して実行できるオペレーションを、コンパイラに対して記述している。一般的に、アンボックス化値型については、型情報は出力コードに出力されない。型"int16"は、本発明の実施形態における組み込み値型の例である。値型に関する既述の説明はユーザ定義の値型にも、組み込み型にも適用できるので、ランタイム時に効率的に処理することができる。コンパイラが、ある特定の値型、特にユーザ定義の値型のメタデータにアクセスできるようになっていなければ、ユーザはソースコードファイルまたはコンフィギュレーション（構成）ファイルの中でメタデータを与えることができる。

【００４４】

データ型のリスト例は図３に示されている。このリストには、組み込み値型のグループ３０２とユーザ定義値型のグループ３０４が含まれている。ユーザ定義値型３０４には、ほとんどすべて種類のデータ構造を含めることができる。ほとんどのソース言語では、ユーザは、組み込み型の組み合わせを利用することによってユーザ定義の値型を作成することができる。この作成は、例えば、型名、その型に含まれる各フィールドのフィールド名、および各フィールドのフィールドタイプを定義することによって行われている。この図示の例では、pointデータ型３０６は２値データ型であり、２次元空間の点（point）のデカルト座標を定義している。circleデータ型３０８は２値データ型であり、円の中心点を定義するpointデータ型の値と、円の半径の大きさを定義するintegerデータ型の別の値を含んでいる。rectangleデータ型は４値データ型であり、矩形の４隅の各々に対するpointデータ型の値を含んでいる。特に、circleデータ型３０８とrectangleデータ型３１０はpointデータ型３０６を利用できる。従って、circleデータ型３０８とrectangleデータ型３１０はpointデータ型３０６から「継承」していると言うことができる。これが真の継承であるとする実装もあれば、ある値型は他の値型を作るために利用されるにすぎないとする実装もある。

【００４５】

ユーザは、ボックス化形式の値型を記述するためにメタデータを作成することができる。例えば、ユーザ定義の値型を作成するプロセスには、値型をボックス化するために必要とされる、その型のメタデータを指定するステップを含めることができる。代表例として、ユーザ定義の型３０４を定義するメタデータはビット列（つまり、値）を記述しており、その中には、型名、その型に含まれるすべてのフィールドのフィールド名、すべてのフィ

ールドのフィールドタイプ、およびその型に関連して実行できるオペレーション（つまり、メソッド）が含まれている。ユーザ定義の型を定義するメタデータには、その型がボックス化形式で実装しているインタフェースのリストを含めることもできる。最後に、メタデータは、その型のボックス化バージョンがオブジェクトクラス階層１１６内のどこに収められるかを示す標識を含むことができる。これについては、図４を参照して以下で説明する。このメタデータは、型の安全性を検証し、値型のボックス化バージョンを管理するためにコンパイラ、ローダおよび/またはランタイム環境によって使用することができる。

【００４６】

図４は、オブジェクトクラス階層３５０の例を示す図である。一般的に、クラス階層３５０のオブジェクトは値型３００よりも複雑なデータ型である。各オブジェクトは、各オブジェクトの型が出力コードの中にその表現で明示的にストアされるという意味で、自己型付け（self-typing）である。オブジェクトは、そのオブジェクトを他のすべてのオブジェクトと区別するID（例えば、オブジェクト名、クラス名）をもっている。また、各オブジェクトは、値（関連の値型をもつ）とそのオブジェクトに関連するメソッドを含む、他のデータをストアするために使用できるフィールド（またはデータメンバ）をもっている。当然のことであるが、オブジェクト内のフィールドは、それ自身がオブジェクトになることができる。オブジェクトは、ロケーション情報（例えば、ポインタ）とインタフェース情報を含むこともできる。クラス階層３５０のような、クラス階層のオブジェクトは、基底ルートオブジェクト（base root object）から派生しているのが代表的である。図４に示すように、この基底ルートオブジェクトはBaseObject３２０で示されている。従って、他のオブジェクトはオブジェクトクラス階層３５０内のBaseObject３２０の下に示されているので、BaseObject３２０から継承している。

【００４７】

図４のオブジェクトクラス階層は、本発明の一形態では値型の二重表現を示している。図４は代表的なクラス階層を示し、そこには、図３に示す値型のボックス化表現が含まれている。オブジェクトクラス階層３５０では、組み込み値型３５２（例えば、integer３２５、floats３２６、およびBoolean３２８）およびユーザ定義の値型３５４（例えば、point３３０、rectangle３３２、circle３３４）は、オブジェクトクラス階層内に他の任意のオブジェクトしてストアされている。図４に示す組み込み値型３５２とユーザ定義の値型３５４はボックス化値型である。従って、組み込み値型３５２とユーザ定義の値型３５４は、階層内の、他の任意のオブジェクトと同じ基準でランタイム時に処理することができる。上述したように、ボックス化値型は、アンボックス化値型をメタデータと関連付け、オブジェクトライクの属性をもつボックス化値型を得ることによって、アンボックス化値型から作成される。メタデータは、図５を参照して以下に詳しく説明されている。

【００４８】

特に、Object yのような、「子(child)」オブジェクトは、Object xのような、「親(parent)」の属性を継承している。例えば、メソッドがObject xに関連付けられていれば、そのメソッドは継承によってObject yにも関連付けられる。図４に示す一形態では、値型のボックス化表現は、値型に親子関係の考え方がない場合でも、親子関係を含むことが可能になっている。例えば、図４において、circleとrectangleは共にpointから派生している。同様に、子のボックス化値型（例えば、circle値型３３４）は、親のボックス化値型（例えば、point値型３３０）からメソッドと他の属性を継承している。このように継承されたメソッドは仮想メソッド(virtual method) と呼ばれている。本発明では、値型とオブジェクト型の二重表現になっているので、開発者は、どちらの形体がメソッドに渡されるかについて気にする必要がない。従って、アンボックス化値型を、ボックス化表現が期待されているオブジェクトメソッドに渡す場合があり、その逆の場合も同じである。コンパイラおよび/またはランタイムは、特定の実装に該当するものとしてコンパイル時にも、ランタイム時にも、該当する表現を選択することができる。

【００４９】

値型をオブジェクトクラス階層の中でアンボックス化値型とボックス化値型として二重表現するということは、値型がメソッドをもつことができ、ある場合にはオブジェクトとして振る舞い、他の場合にはアンボックス化値型として振る舞うことができることを意味している。実際には、値型がボックス化表現にあるとき、値型は他のオブジェクトと同じようにメソッドをもつことができるという効果が得られる。値型がアンボックス化表現にあるときは、メソッドがオブジェクト型（例えば、ボックス化表現）を予想している場合でも、有効な引数としてメソッドに渡すことができるという効果が得られる。ボックス化値型表現とアンボックス化値型表現の両方が利用できるので、開発者は、どちらのバージョンを使用するか、一方の形体から他方の形体への変換といったことを明示的に指定しなくても、この機能を得ることができる。

【００５０】

データ型による値の記述は、その値の表現と、その値に対して実行できるオペレーションが完全に定義されている場合に行われる。データ型では、値の表現の定義は、その値の表現を構成するビット列を記述することによって行われる。データ型に対して実行できるオペレーション群の定義は、オペレーションごとに名前付きメソッド（named method）を指定することによって行われる。名前付きメソッドは、データ型に関連して実行できるオペレーションを記述している。

【００５１】

オブジェクトについては、オブジェクトの表現の定義は、そのオブジェクトのロケーションと、オブジェクトの表現を構成するビット列を記述することによって行われる。従って、オブジェクトには、オブジェクトのコンテンツ（内容）の定義と、そのオブジェクトに対して実行できるオペレーションが含まれている。あるオブジェクトが値を含んでいるとき、その定義には、その値の表現と、その値に関連して有効に実行できるオペレーション（例えば、メソッド）が含まれている。オブジェクトの定義は、値の表現を構成するビット列（自己記述データ（self-describing data））、オブジェクトのロケーション（ポインタデータ）、およびオブジェクトに対する少なくとも１つの名前付きメソッド（インタフェースデータ）を記述することによって行われる。

【００５２】

従って、オブジェクトとアンボックス化データ型の違いは、オブジェクトが生データ（つまり、値表現）だけでなく、オブジェクトのロケーションを含む他のデータも含んでいる点にある。上記の他のデータは、メタデータとしてオブジェクトにストアされている。メタデータは、どの特定プログラミング言語からも独立した形でストアできるという利点がある。従って、メタデータは、オブジェクトを操作するツール（例えば、コンパイラやデバッガ）の間で共通にやりとりするメカニズムとして利用することができる。

【００５３】

次に図５を参照して説明すると、アンボックス化値型４００は、図示のように、生の値データ４０１（つまり、値表現）だけを含んでいる。ボックス化値型４０２は、図示のように、生の値データ４０１だけでなく、メタデータ４０４も含んでいる。値型（組み込みまたはユーザ定義）ごとに、対応するボックス化値型を作成することができる。ボックス化データ型が、上述したようにオブジェクトの特性をもっているのは、ボックス化データ型は、メタデータを通して値記述データ、ロケーションデータ、およびメソッドデータと関連付けられるからである。従って、メタデータはボックス化値型と関連付けられているので、ボックス化値型は図４のオブジェクトクラス階層３２０にストアしておくことができる。本発明の例示の実施形態では、図５に示すボックス化値型とアンボックス化値型は、出力コード１０８（図１）にストアすることができる。当然に理解されるように、図５はボックス化値型の論理的表現を示し、メタデータが値型のストレージロケーション（記憶場所）と関連付けられていることを示している。

【００５４】

本発明の形態を取り入れている別のシステム５００の機能ソフトウェアコンポーネントは図６に示されている。システム５００には、コンパイラ５０２、５０４および５０６のよ

うに、少なくとも１つのフロントエンドコンパイラが組み入れられているが、これは本発明の要求条件を示すのではなく、複数のまたは結合されたフロントエンドシステムに適用される本発明の考え方を示すことだけを目的としている。フロントエンドコンパイラ５０２、５０４および５０６は、ソースファイル５０８、５１０および５１２のように、異種タイプのソース言語ファイルを、それぞれ構文解析（parse）し、分析する機能を備えている。これらのソースファイル５０８、５１０および５１２には、組み込み値型、ユーザ定義の値型、およびオブジェクトを含めることができる。この実施形態では、フロントエンドコンパイラ５０２、５０４および５０６は、各々が共通言語出力ファイル５１４、５１６および５１８を生成する。一般的に、コンパイラ５０２、５０４および５０６は、図１を参照して上述したコンパイラ１０４と機能的に類似している。

【００５５】

本発明の例示の実施形態では、共通言語出力ファイル５１４、５１６および５１８は、異種タイプの、複数のソース言語、例えば、手続き型、機能型およびオブジェクト指向プログラミングの概念を表現するのに適している「共通」（汎用的であるという意味である）中間言語の実行可能命令をもっているので、どのソース言語が使用されるかに関係なく、１つのタイプの中間言語を使用するだけで済むことになる。共通言語出力ファイル５１４、５１６および５１８内の実行可能命令は、プロセッサによって直接に実行可能な命令（例えば、オブジェクトまたはネイティブマシンコード）にすることも、あるタイプの実行環境内で実行される「中間」型命令（例えば、Java(登録商標)のバイトコード、pコード、その他の中間言語）にすることも可能である。

【００５６】

フロントエンドコンパイラ５０２、５０４および５０６は、それぞれのソースファイル５０８、５１０および５１２を読み取って、解析する機能を備えているほかに、共通言語で表されたファイルを読み取って、解析する機能も備えている。さらに、共通言語で表された関数のライブラリ宣言ファイル５２０が用意されているので、フロントエンドコンパイラ５０２、５０４および５０６で利用可能になっている。

【００５７】

共通言語ファイル５１４、５１６および５１８は、コンパイルされたあと、実行環境またはランタイム環境５２２に送ることができる。本明細書では、実行環境とランタイム環境は同じ意味で使用されている。実行環境は、直接実行環境にすることも、管理されたランタイム環境にすることも、管理されていないランタイム環境にすることもできる。アンボックス化値型をボックス化値型に変換する（またはその逆に）必要があるとき、その変換をコンパイルステージで行うと、ランタイム環境の状況が管理されているか、管理されていないかに関係なく、変換された値型を使用できるという利点があるが、ランタイム環境で行うことも可能である。実際には、環境は、コンパイルされたファイルを読み取り、実行できる環境ならば、どのタイプの環境であっても構わない。図６に示すランタイム環境５２２は、以下で説明するように、複数の特徴、関数およびサービスをもつ、管理された環境を示している。

【００５８】

ランタイム環境５２２に渡される前に、各出力ファイル５１４、５１６および５１８は、図示のようにオプションの独立処理セクション５２４またはオプションの内蔵処理セクション５２６でオプションの処理を行っておくことも可能である。一般的に、オプションの処理では、検証、型チェック、および/または共通言語をランタイム環境５２２で使用するのに適した形に変換することが行われる。従って、オプションの処理は、受け取った共通出力ファイル５１４、５１６および５１８を翻訳（変換）し、解釈し、さもなければ、実行環境５２２で実行可能な出力コードに変換するために使用できる。

【００５９】

実行環境５２２が、図６に示すように管理されたランタイム環境である場合には、ランタイム環境自身がファイルを実行のためにロードするローダ５３０をもっている。ローダ５３０は実行可能ファイルを受け取り、必要な参照を解決した後、コードをロードする。

この環境には、スタックウォーカ（stack walker）５３４、すなわち、メソッドコールを管理し、任意の時点にスタックに置かれているメソッドコールのシーケンスを特定できるようにするコード部分が用意されていることがある。レイアウトエンジン（layout engine）５３２が用意されていることもあり、これは、種々のオブジェクトと他のエレメントの、メモリ内のレイアウトを実行されるアプリケーションの一部として設定する。さらに、実行環境には、あるコードがあるシステムリソースにアクセスする許可（あるいは、いやしくも実行する許可）をもっているかどうかを判断することによって、リソースの無許可使用を防止するセキュリティモジュール５３８が用意されていることもある。ランタイム環境は、ガーベッジコレクタ（garbage collector）５３６のようなメモリ管理サービスと、デバッガとプロファイリング（profiling）のような、他の開発サービス５４０を備えていることもある。管理された実行環境によって提供できる、他のタイプのサービスとしては、特に、コードを検証してから実行されるようにするサービスがある。

【００６０】

実行環境５２２は、さらに、共通ライブラリプログラムファイル５２８を利用することが可能であるが、そこには、共通ライブラリ宣言５２０の機能を実行するための実際の実装情報が入っている。

【００６１】

ランタイム時に、出力ファイル５１４、５１６および５１８はランタイム環境５２２にロードされる。重要なことは、図５に示すボックス化またはアンボックス化値型のように、ランタイム環境に渡される情報は、ランタイムに先立ってオブジェクトを形成するためにランタイム環境によって使用されることである。レイアウトエンジンは、該当のメソッドとフィールド情報を含んでいるデータ構造を、クラスのタイプ別に作成するためにこの情報を使用するのが一般である。

【００６２】

図７は、本発明の例示の実施形態において個々の値型をボックス化し、アンボックス化するときのオペレーションの流れ（フロー）を示す図である。ボックス化とアンボックス化は自動的に行うことができるので、特定データ型のどちらのバージョンも、ランタイム時に常に利用可能にしておくことができる。従って、状況（例えば、アンボックス化値型をオブジェクトに割り当てる）に応じて、最も効率的な形体の値型を選択的に使用することができる。当然のことであるが、コンパイラ１０４が、変換された形体の値型は不要であると判断した場合には、変換を回避することも可能である。

【００６３】

図７の論理的オペレーションは、(1) コンピュータシステム上で実行されるコンピュータ実装ステップのシーケンスまたはプログラムモジュールとして、および/または (2) コンピュータシステム内の相互接続ロジック回路またはマシンロジックモジュールとして実装されている。どのような実装にするかは選択の問題であり、本発明を実現するコンピュータシステムに要求されるパフォーマンスによって決まる。従って、本明細書で説明している本発明の実施形態を構成する論理的オペレーションは、オペレーションとも、ステップとも、モジュールとも呼ばれる。この分野の精通者ならば理解されるように、これらのオペレーション、ステップおよびモジュールは、特許請求の範囲に記載されている本発明の精神と範囲から逸脱しない限り、ソフトウェアでも、ファームウェアでも、特殊目的デジタルロジックでも、あるいはこれらの任意の組み合わせでも、実現することが可能である。

【００６４】

本発明の一実施形態では、図７のオペレーションはステップ６００からスタートし、アンボックス化テストオペレーション６０２に進む。テストオペレーション６０２は、アンボックス化からボックス化への変換を要求するトリガが、ソースファイルに含まれているかどうかを検出する。アンボックス化からボックス化への変換トリガ（unboxed to boxed conversion trigger）は、ソースファイル内のエンティティで、アンボックス化からボックス化への変換が必要であると示しているものがトリガとなる。アンボックス化からボッ

クス化への変換の例としては、アンボックス化値型がボックス化値型に割り当てられる場合や、ボックス化値型または別のオブジェクトを予想しているオブジェクトにアンボックス化値型が渡される場合がある。どちらの例の場合も、アンボックス化からボックス化への変換が必要になる。アンボックス化からボックス化への変換を要求するトリガが、ソースファイルに含まれていることがテストオペレーション６０２で検出されると、オペレーションフローはYESにブランチして、アンボックス化値型出力オペレーション６０４に移る。アンボックス化値型出力オペレーション６０４は、アンボックス化からボックス化への変換を実行するためのコンパイラからのコード、アンボックス化値型、およびアンボックス化値型に関連するメタデータを、出力コードを通してランタイム環境に出力する。次に、オペレーションフローは変換オペレーション６０８に移り、そこで、出力コード、アンボックス化値型、およびそのアンボックス化値型に関連するメタデータから、ボックス化値型がランタイム時に変換または構築される。オペレーションフローはオペレーション６１９に進み、変換された値型に対して終了する。

【００６５】

ボックス化値型は、型名の定義、フィールド名、フィールドタイプ、およびボックス化値型に関連して実行できるオペレーション（例えば、メソッド）を含むことが可能である。ボックス化値型の作成には、ボックス化値型がオブジェクトクラス階層に置かれている該当位置および他のボックス化値型との関係を表しているメタデータの作成を含めることもできる。

【００６６】

特に注目すべきことは、アンボックス化からボックス化への変換を要求するトリガがソースファイルに含まれていないと、アンボックス化テストオペレーション６０２で判断された場合にも、ボックス化テストオペレーション６１０に移ることである。ボックス化テストオペレーション６１０は、ボックス化からアンボックス化への変換を要求する変換トリガがソースファイルに含まれているかどうかを検出する。ボックス化からアンボックス化への変換を要求する変換トリガがソースファイルに含まれていなければ、オペレーションフローはNOにブランチしてステップ６１２に移り、変換された値型に対して終了する。

【００６７】

他方、ボックス化からアンボックス化への変換を要求する変換トリガがソースファイルに含まれているとボックス化テストオペレーション６１０で検出されると、オペレーションフローはYESにブランチしてボックス化値型出力オペレーション６１４に移る。ボックス化値型出力オペレーション６１４は、ボックス化からアンボックス化への変換を行うコード、ボックス化値型、およびそのボックス化値型に関連するメタデータを、出力コードを通してランタイム環境に出力する。そのあと、オペレーションフローは変換オペレーション６１６に進み、出力コードに入っている、ボックス化からアンボックス化への変換を実行するコード、ボックス化値型、およびそのボックス化値型に関連するメタデータからアンボックス化値型を変換または作成する。

【００６８】

ボックス化からアンボックス化への変換トリガは、ソースファイル内にあって、変換が必要であることを示しているエンティティならば、どのエンティティであっても構わない。変換トリガの例としては、ボックス化値型がアンボックス化値型に割り当てられる場合およびアンボックス化値型を予想しているオブジェクトにボックス化値型が渡される場合がある。どちらの例の場合も、ボックス化からアンボックス化への変換が必要になる。オペレーションフローはステップ６１８に進み、変換された値型に対して終了する。

【００６９】

以上から理解されるように、図７のオペレーションは、ソースファイルの中で見つかった各々の値型に対して、必ずしも変換が行われないように変更することができる。変換が適切かどうかの判断は、予備段階で行われるようにすることもできる。例えば、ボックス化組み込み値型は、ソースファイルではアンボックス化フォーマットで書かれることはないとコンパイラが認識する場合がある。そのような場合には、変換は、不要であるものとし

て回避されることになる。

【００７０】

上述したように、図７に関連して説明したプロセスは、個別的値型を論理的に処理することに関係している。しかし、代表例としては、複数の値型がソースファイルに含まれているので、ボックス化かアンボックス化のどちらかが必要になる場合がある。本発明の一実施形態では、検出オペレーション６０２と６１０および出力オペレーション６０４と６１４は、コンパイルステージで複数の値型に対して実行されてから、ランタイム時の変換オペレーション６０８と６１６に移るようになっているのが一般である。このようにすると、必要とされる変換コードは、その大部分（または全部）が出力コードに出力され、ランタイム時に実行されることになる。

【００７１】

しかるに、本発明の代替実施形態では、検出オペレーション６０２と６１０および出力オペレーション６０４と６１４は、ランタイム時にも実行されるようになっている。「出力」オペレーションは、ランタイム時のボックス化またはアンボックス化コードへのコールによって実現されている。このような実施形態としては、例えば、図８と図９に示すものがある。

【００７２】

オブジェクトコード１０６（図１）の実行時に、ランタイム環境５２２（図６）のようなランタイム環境は、仮想メソッドの実装の場合と同じように、特定データ型のボックス化バージョンを使用するか、アンボックス化バージョンを使用するかを判断することができる。一実施形態では、フロー７００はある特定の場合を示し、そこでは、コンパイラとは対照的に、ランタイム環境が選択オペレーションを実行するようになっている。初期には、フローは、定義オペレーション７０２からスタートするのが一般であり、そこで、関数コールの中で使用される特定の値型が定義される。値型の定義は、その値型がボックス化であるか、アンボックス化であるかといった、ある種の情報を与えることによって行われるのが一般である。一実施形態では、値型にビットを関連付けることが可能であり、そのビットは、値型がボックス化であるか、アンボックス化であるかに応じてセットまたはクリアされるようになっている。

【００７３】

値型が定義されると、パス（引き渡し）オペレーション７０４は、定義された値型を特定の関数に引き渡す。基本的には、コンパイル時に、コンパイラは、ランタイム時に値型を関数に渡せるようにする出力コードを出力している。この種のパラメータを関数に渡すことは直接的であり、パラメータを渡すことにより、必要な値型情報が関数に与えられてオペレーションが行われることになる。

【００７４】

定義された値型を受け取る特定の関数は、ボックス化かアンボックス化のどちらかの値型を予想している。従って、パスオペレーション７０４に続いて、判断オペレーション７０６は、渡された値型が、予想の値型と同じであるかどうかを判断する。この判断は、値型に関連するビットを単純にテストするだけにすることも、関数に渡されたデータの型を別のオペレーションで評価するようにすることも可能である。

【００７５】

渡された値型が予想の値型と同じでないと判断オペレーション７０６で判断されると、フローはNOにブランチして変更 (modify) オペレーション７０８に移る。変更オペレーションは渡された値型をボックス化するか、アンボックス化し、新しい値型を関数に引き渡す。関数がボックス化値型を予想していたが、受け取った値型がアンボックス化であるような実施形態では、ステップ７０８は値型をボックス化する。他方、関数がアンボックス化値型を予想していたが、受け取った値型がボックス化であれば、オペレーション７０８は値型をアンボックス化することになる。ボックスオペレーション７０８の結果は、アンボックス化値型を記述しているオブジェクトを指しているポインタである。変更オペレーション７０８に続いて、フローはコールオペレーション７１０へ進む。このオペレーション

は以下で説明する。

【００７６】

渡された値型が予想の値型と同じであると判断ステップ７０６で判断されたときは、フローはYESにブランチしてコールオペレーション７１０に移る。基本的には、関数が受け取った値型が予想していた通りであれば、変更オペレーション７０８のような変更は不要になる。従って、フローはコールオペレーション７１０を続ける。

【００７７】

コールオペレーション７１０は、定義された関数に関連付けられたメンバメソッドをコールする。例えば、スーパクラスの場合や、関数がボックス化とアンボックス化の両バージョンの値型を受け取る可能性のある場合のように、コンパイラがコンパイル時にどのメソッドをコールすべきか判断できず、従って、複数のバージョンが実装されている場合には、メソッドを仮想メソッドにすることができる。メンバメソッドは、渡された値型に対してオペレーションを行う、ユーザ定義のメソッドと関係付けられている。コンパイル時に、コンパイラは関数内で種々のメソッドを実行するためのコードを出力する。しかし、関数は異なる型を受け取ることがあるので、コンパイラは、各々の型ごとに特定のコードを関数内に挿入することはしない。その代わり、コンパイラは仮想メソッドテーブル (virtual method table) を作成する。仮想メソッドテーブルには、関数が受け取る可能性のある、特定の値型の各々に対してメソッドを実行するために必要な情報が収められている。

【００７８】

従って、メソッドが７１０でコールされると、ランタイム環境は、仮想メソッドテーブル内にある該当メソッドへの参照を使用してメソッドコールを実行する。さらに具体的には、ルックアップオペレーション７１２は、渡された値型に関係する特定メソッドを探し出す。その型に対する特定メソッドが見つかると、実行オペレーション７１４はその値型に対するメソッドを実行する。

【００７９】

すでに述べたように、上述したオペレーションのフローは、ボックス化とアンボックス化のどちらのパラメータでも処理することができる。重要な特徴は、特定の関数に渡すことができる、異種の値型に対するメソッドを得るために仮想テーブルを使用していることである。関数は異種の型を受け取ることがあるため、またコンパイラは関数がどの型を受け取るかを知らないので、ランタイム環境は必要な解析を実行し、値型の間に不一致があればそれを解決する。

【００８０】

図８に代わる代替実施形態のオペレーションフロー８００は図９に示されている。フロー８００の最初の２オペレーションである、定義オペレーション８０２とパスオペレーション８０４は、図８を参照して説明したオペレーション７０２および７０４に類似している。つまり、定義オペレーション８０２は値型をボックス化またはアンボックス化として定義し、パスオペレーション８０４はその値を関数に渡している。

【００８１】

値が渡されると、コールオペレーション８０６はメンバ関数をコールする。このオペレーションは上述したコールオペレーション７１０に類似し、実際のメソッドがコールされる。コールオペレーション８０６に続いて、ルックアップオペレーション８０８は、渡された値型に対する、コールされたメソッドを仮想メソッドテーブルから探し出す。ルックアップオペレーション８０８は、上述したオペレーション７１２に類似している。

【００８２】

コールオペレーション８０６とルックアップオペレーション８０８に続いて、判断オペレーション８１０は、渡された値型が予想の値型と同じであるかどうかを判断する。判断オペレーション８１０は、渡された値型が予想の値型形式と突き合わせて解析される点で、上述した判断オペレーション７０６に類似している。ただし、１つの違いは、判断オペレーションを実行するコードが、実際には、以下で説明するように、コールされたメソッド

コードの先頭部分に置かれていることである。

【００８３】

渡された値型が予想形式と異なっていると判断オペレーション８１０で判断されると、フローはNOにブランチして変更オペレーション８１２に移る。変更オペレーション８１２は、図８を参照して上述した変更オペレーション７０８に類似している。基本的に、変更が必要であれば、変更オペレーション８１２は、値型を必要に応じてボックス化またはアンボックス化するために必要なオペレーションを実行する（例えば、該当するボックス化またはアンボックス化コードをコールする）。変更が行われると、実行オペレーション８１４は、変更された値型を使用してメソッドを実行する。

【００８４】

他方、渡された値型が予想形式と同じであると判断オペレーション８１０で判断されると、実行オペレーション８１４は、渡された値型を使用してメソッドを実行する。渡された値型は予想の値型と同じであったので、オペレーション８１２のような変更オペレーションは、実行前に必要でない。

【００８５】

図９に示す実施形態によれば、コーラ（呼び出し側）オブジェクトは、メソッドをコールし、そのメソッドを実行する関数に値型を渡すだけで済むことになる。コーラは、値型が正しいかどうかの判断を行わないで済むことになる。このようなコーラオブジェクトはストリームライン化されているので、実行するオペレーションは少なくなっている。しかし、判断オペレーションはメソッドまたは他のなんらかのモジュールに実行させる必要があるとのトレードオフがある。基本的には、プラミング（plumbing）は、メソッドの実行に先立って実行されるコードの小さな部分として置いておくことができる。コーラの数がコールされるメソッドの数より多いときは、コーラオブジェクトのストリームライン化を利用すると、便利である。

【００８６】

以上のように、本発明は、種々のプログラミング言語のオブジェクトを処理し、ユーザ定義のデータ型をボックス化し、アンボックス化するために、メソッドとしても、装置としても、あるいはコンピュータプログラムを収容しているコンピュータ読み取り可能媒体やプログラムプロダクトのような製造物品としても実現することが可能である。本発明の好ましい実施形態に関連して、本発明を具体的に示し、説明してきたが、この分野の精通者ならば理解されるように、本発明の精神と範囲から逸脱しない限り、その形体と細部において種々態様に変更することが可能である。

【図面の簡単な説明】

【図１】本発明の実施形態による型システムを統一化された見方で提供するように動作する、例示コンパイラの論理的表現を示す図である。

【図２】本発明の例示実施形態のための動作環境を提供するコンピュータシステムを示す図である。

【図３】データ型をカテゴリ化するために本発明の例示実施形態で使用される例示データ型を示す図である。

【図４】オブジェクトを編成するために本発明の例示実施形態で使用される例示オブジェクトクラス階層を示す図である。

【図５】ボックス化データ型とアンボックス化データ型のセットを示す図である。

【図６】図１の例示コンパイラシステムの論理的表現を示す詳細図である。

【図７】本発明の例示実施形態において値型をボックス化し、アンボックス化するための方法を示す図である。

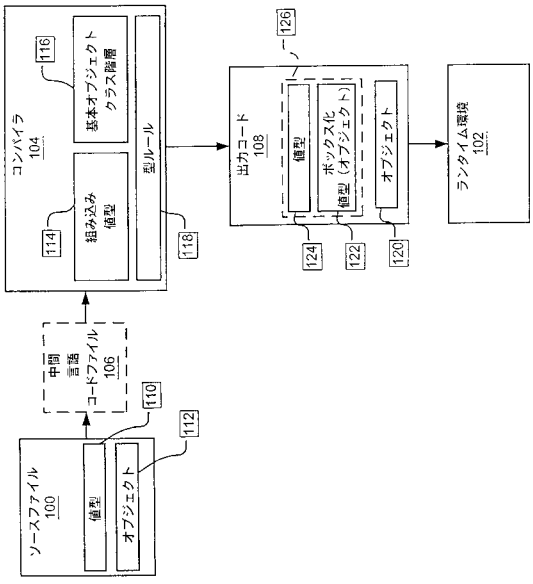
【図８】本発明の実施形態においてランタイムに値型のボックス化とアンボックス化を実現するための方法を示す図である。

【図９】本発明の他の実施形態においてランタイムに値型のボックス化とアンボックス化を実現するための方法を示す図である。

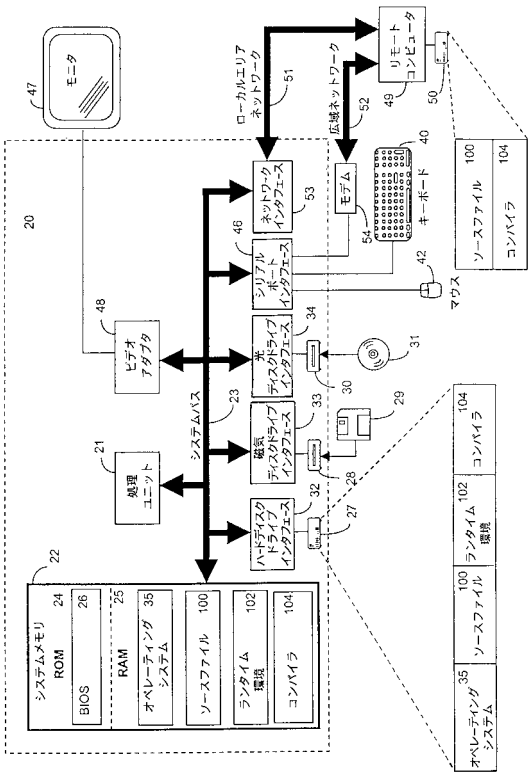
【符号の説明】

１００　ソースファイル
１０２　ランタイム環境
１０４　コンパイラ
１０６　中間言語コードファイル
１０８　出力コード
１１０　値型
１１２　オブジェクト
１１４　組み込み値型
１１６　基本オブジェクトクラス階層
１１８　型ルール
１２０　オブジェクト
１２２　ボックス化値型
１２４　値型
３０２　組み込み型
３０４　ユーザ定義の型

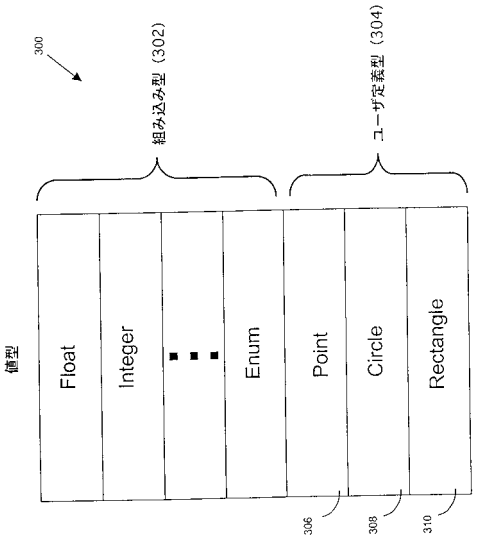
【図１】



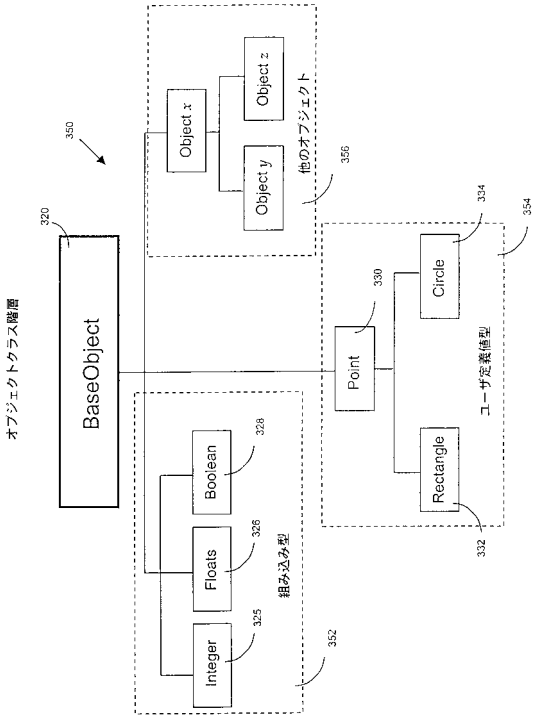
【図２】



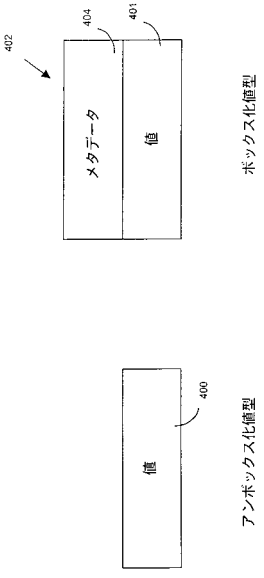
【図３】


300
組み込み型（302）
ユーザ定義型（304）
値型
Float
Integer
Enum
Point
Circle
Rectangle
306
308
310

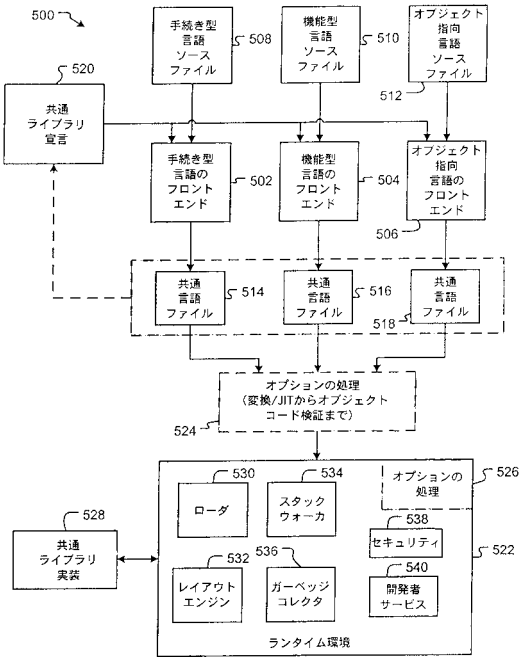
【図４】


350
オブジェクトクラス階層
BaseObject
320
他のオブジェクト
356
Object x
Object y
Object z
ユーザ定義値型
354
330
Point
334
Circle
332
Rectangle
組み込み型
352
328
Boolean
326
Floats
325
Integer

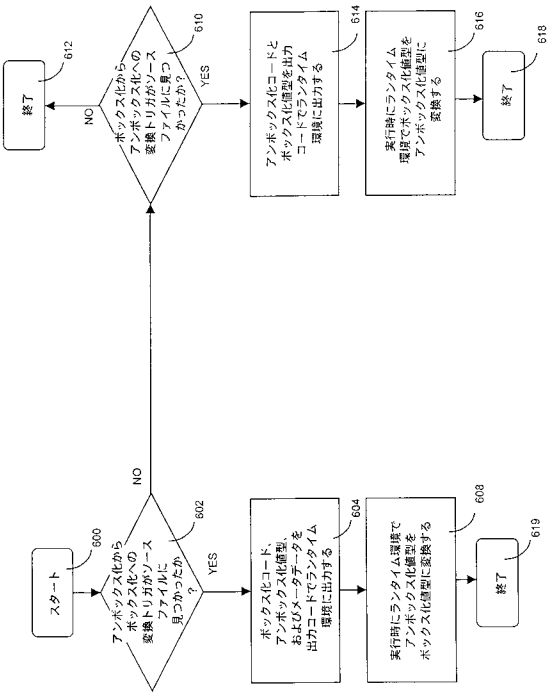
【図５】


402
404
401
メタデータ
値
ボックス化値型
値
400
アンボックス化値型

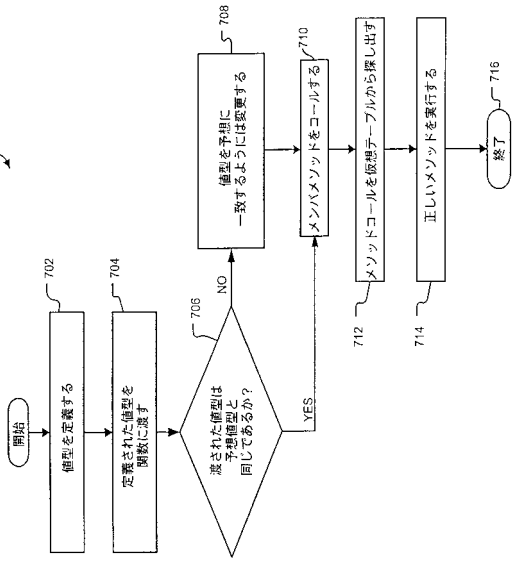
【図６】


500
手続き型言語ソースファイル
508
機能型言語ソースファイル
510
オブジェクト指向言語ソースファイル
512
520
共通ライブラリ宣言
手続き型言語のフロントエンド
502
機能型言語のフロントエンド
504
オブジェクト指向言語のフロントエンド
506
共通言語ファイル
514
共通言語ファイル
516
共通言語ファイル
518
オプションの処理（変換/JITからオブジェクトコード検証まで）
524
ローダ
530
スタックウォーカ
534
オプションの処理
526
528
共通ライブラリ実装
セキュリティ
538
レイアウトエンジン
532
ガーベッジコレクタ
536
開発者サービス
540
522
ランタイム環境

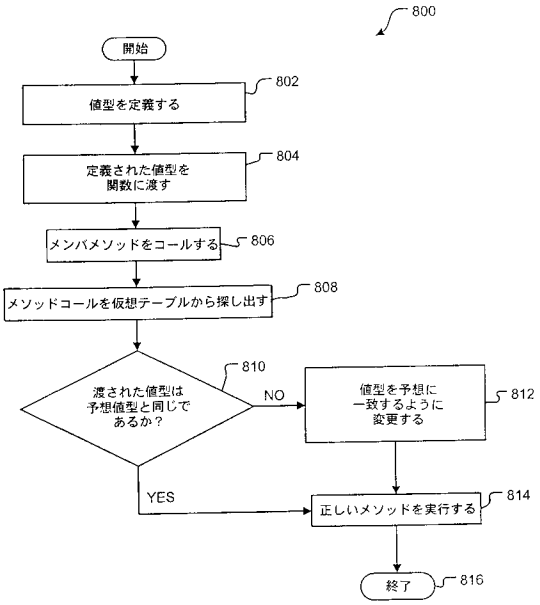
【図７】


スタート
600
アンボックス化から
ボックス化への
変換トリガがソース
ファイルに
見つかったか
？
602
NO
YES
ボックス化コード、
アンボックス化値型、
およびメータデータを
出力コードでランタイム
環境に出力する
604
実行時にランタイム環境で
アンボックス化値型を
ボックス化値型に変換する
608
終了
619
ボックス化から
アンボックス化への
変換トリガがソース
ファイルに見つかったか？
610
NO
YES
終了
612
アンボックス化コードと
ボックス化値型を出力
コードでランタイム
環境に出力する
614
実行時にランタイム
環境でボックス化値型を
アンボックス化値型に
変換する
616
終了
618

【図８】


700
開始
値型を定義する
702
定義された値型を
関数に渡す
704
渡された値型は
予想値型と
同じであるか？
706
NO
YES
値型を予想に
一致するようには変更する
708
メンバメソッドをコールする
710
メソッドコールを仮想テーブルから探し出す
712
正しいメソッドを実行する
714
終了
716

【図９】


800
開始
値型を定義する
802
定義された値型を
関数に渡す
804
メンバメソッドをコールする
806
メソッドコールを仮想テーブルから探し出す
808
渡された値型は
予想値型と同じで
あるか？
810
NO
YES
値型を予想に
一致するように
変更する
812
正しいメソッドを実行する
814
終了
816

フロントページの続き

(72)発明者 ジョージ エイチ．ボスワース
アメリカ合衆国 ９８０７２ ワシントン州 ウッディンビル ノースイースト １２３ コート １９８３０
(72)発明者 パトリック エイチ．ダサッド
アメリカ合衆国 ９８００６ ワシントン州 ベルビュー １４２ コート サウスイースト ６００８
(72)発明者 ジェームズ エス．ミラー
アメリカ合衆国 ９８００８ ワシントン州 ベルビュー ノースイースト ４ プレイス １７２１３
(72)発明者 ダリル ビー．オーランダー
アメリカ合衆国 ８０３０４ コロラド州 ボールダー ジュニパー アベニュー ７２０

合議体
審判長 吉岡 浩
審判官 石井 茂和
審判官 清木 泰