

(19) 日本国特許庁 (JP)

(12) 特 許 公 報 (B2)

(11) 特許番号

特許第5021805号
(P5021805)

(45) 発行日 平成24年9月12日 (2012.9.12)

(24) 登録日 平成24年6月22日 (2012.6.22)

(51) Int. Cl.

F I

G 0 6 F 9/44 (2006.01)

G 0 6 F 9/44 5 3 0 M

G 0 6 F 9/45 (2006.01)

G 0 6 F 9/44 3 2 2 A

請求項の数 12 (全 25 頁)

(21) 出願番号	特願2010-270690 (P2010-270690)	(73) 特許権者	500046438
(22) 出願日	平成22年12月3日 (2010.12.3)		マイクロソフト コーポレーション
(62) 分割の表示	特願2007-287939 (P2007-287939) の分割		アメリカ合衆国 ワシントン州 9805 2-6399 レッドモンド ワン マイ クロソフト ウェイ
原出願日	平成13年7月10日 (2001.7.10)	(74) 代理人	110001243
(65) 公開番号	特開2011-70696 (P2011-70696A)		特許業務法人 谷・阿部特許事務所
(43) 公開日	平成23年4月7日 (2011.4.7)	(72) 発明者	ジョージ エイチ. ボスワース
審査請求日	平成22年12月3日 (2010.12.3)		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 1 23 コート 19830
(31) 優先権主張番号	09/613289	(72) 発明者	パトリック エイチ. ダサッド
(32) 優先日	平成12年7月10日 (2000.7.10)		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
(33) 優先権主張国	米国 (US)		最終頁に続く
(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)		

(54) 【発明の名称】 データ型を処理するシステムおよび方法

(57) 【特許請求の範囲】

【請求項 1】

処理ユニットと、前記処理ユニットにシステムバスを介して結合されたシステムメモリとを備えた、データ型を処理するシステムであって、

前記システムメモリは、値型として定義されたデータおよびオブジェクトとして定義されたデータを含むソースファイルと、ランタイム環境と、コンパイラとを格納し、

前記値型として定義されたデータは値のみを含み、前記オブジェクトとして定義されたデータは値および該値を記述するメタデータを含み、

前記コンパイラは、前記処理ユニットに対し、前記ソースファイルを前記処理ユニットで実行されるランタイム環境により実行可能な出力コードに変換させ、

前記コンパイラは、前記処理ユニットに対し、前記ソースファイルを解析させ、かつ前記オブジェクトとして定義されたデータが、アンボックス化値型表現を受け取ることを予想する関数に引き渡されることを判断させ、前記コンパイラは前記処理ユニットに対し自動的に、前記ソースファイルでオブジェクトとして定義されたデータから前記メタデータを取り除いて前記オブジェクトのアンボックス化値型表現を生成して前記出力コード内にアンボックス化値型表現を生成させ、

前記ランタイム環境を実行中の前記処理ユニットが前記出力コードを解析し、前記オブジェクトとして定義されたデータがアンボックス化値型表現を受け取ることを予想する関数に引き渡されると判断した場合、前記ランタイム環境は、前記オブジェクトとして定義されたデータに代えて、前記アンボックス化値型表現のデータを前記関数に引き渡すこと

10

20

を特徴とするシステム。

【請求項 2】

前記オブジェクトはユーザにより定義されたデータ型であり、前記メタデータは、前記データ型の型名、前記データ型に含まれるフィールドのフィールド名、前記フィールドのフィールドタイプ、および前記データ型に関して実行できるオペレーションを含むことを特徴とする請求項 1 に記載のシステム。

【請求項 3】

前記メタデータは、前記オブジェクトに含まれた値がどのように解釈されるかを記述することを特徴とする請求項 1 または 2 に記載のシステム。

【請求項 4】

処理ユニットと、前記処理ユニットにシステムバスを介して結合されたシステムメモリであって、値型として定義されたデータおよびオブジェクトとして定義されたデータを含むソースファイルと、ランタイム環境と、コンパイラとを格納するシステムメモリとを備えた、データ型を処理するシステムであって、

前記値型として定義されたデータは値のみを含み、前記オブジェクトとして定義されたデータは値および該値を記述するメタデータを含み、

前記コンパイラは、前記処理ユニットに対し、前記ソースファイルを前記処理ユニットで実行されるランタイム環境により実行可能な出力コードに変換させ、

前記コンパイラは、前記処理ユニットに対し、前記ソースファイルを解析させ、かつ前記値型として定義されたデータが、前記オブジェクトとして定義されたデータを受け取ることを予想する関数に引き渡されることを判断させ、前記コンパイラは前記処理ユニットに対し自動的に、前記ソースファイルの値に前記メタデータを追加して前記オブジェクトとして定義されたデータを生成して前記出力コード内にオブジェクトとして定義されたデータを生成させ、

前記ランタイム環境を実行中の前記処理ユニットが前記出力コードを解析し、前記値として定義されたデータが前記オブジェクトとして定義されたデータを受け取ることを予想する関数に引き渡されると判断した場合、前記ランタイム環境は、前記値型として定義されたデータに代えて、前記オブジェクトとして定義されたデータを前記関数に引き渡すことを特徴とするシステム。

【請求項 5】

前記オブジェクトはユーザにより定義されたデータ型であり、前記メタデータは、前記データ型の型名、前記データ型に含まれるフィールドのフィールド名、前記フィールドのフィールドタイプ、および前記データ型に関して実行できるオペレーションを含むことを特徴とする請求項 4 に記載のシステム。

【請求項 6】

前記メタデータは、前記オブジェクトに含まれた値がどのように解釈されるかを記述することを特徴とする請求項 4 または 5 に記載のシステム。

【請求項 7】

データ型を処理する方法であって、前記方法は、

値型として定義されたデータおよびオブジェクトとして定義されたデータを含むソースファイルと、ランタイム環境と、コンパイラとをメモリに格納するステップであって、前記値型として定義されたデータは値のみを含み、前記オブジェクトとして定義されたデータは値および該値を記述するメタデータを含む、メモリに格納するステップと、

前記メモリに結合された前記処理ユニットにより、前記ソースファイルを前記処理ユニットで実行されるランタイム環境により実行可能な出力コードに変換するステップであって、前記コンパイラは、前記処理ユニットに対し、前記ソースファイルを解析させ、かつ前記オブジェクトとして定義されたデータが、アンボックス化値型表現を受け取ることを予想する関数に引き渡されることを判断させ、前記コンパイラは前記処理ユニットに対し自動的に、前記ソースファイルで前記オブジェクトとして定義されたデータから前記メタデータを取り除いて前記オブジェクトのアンボックス化値型表現を生成して前記出力コー

10

20

30

40

50

ド内にアンボックス値型表現を生成させるステップと、

前記処理ユニットが前記出力コードを解析し、前記オブジェクトとして定義されたデータがアンボックス化値型表現を受け取ることを予想する関数に引き渡されると判断するステップと、

前記処理ユニットが、前記オブジェクトとして定義されたデータに代えて、前記アンボックス化値型表現のデータを前記関数に引き渡すステップと

を含むことを特徴とする方法。

【請求項 8】

前記オブジェクトはユーザにより定義されたデータ型であり、前記メタデータは、前記データ型の型名、前記データ型に含まれるフィールドのフィールド名、前記フィールドのフィールドタイプ、および前記データ型に関して実行できるオペレーションを含むことを特徴とする請求項 7 に記載の方法。

10

【請求項 9】

前記メタデータは、前記オブジェクトに含まれた値がどのように解釈されるかを記述することを特徴とする請求項 7 または 8 に記載の方法。

【請求項 10】

データ型を処理する方法であって、前記方法は、

値型として定義されたデータおよびオブジェクトとして定義されたデータを含むソースファイルと、ランタイム環境と、コンパイラとをメモリに格納するステップであって、前記値型として定義されたデータは値のみを含み、前記オブジェクトとして定義されたデータは値および該値を記述するメタデータを含む、メモリに格納するステップと、

20

前記メモリに結合された前記処理ユニットにより、前記ソースファイルを前記処理ユニットで実行されるランタイム環境により実行可能な出力コードに変換するステップであって、前記コンパイラは前記処理ユニットに対し、前記ソースファイルを解析させ、かつ前記値型として定義されたデータが、前記オブジェクトとして定義されたデータを受け取ることを予想する関数に引き渡されることを判断させ、前記コンパイラは前記処理ユニットに対し自動的に、前記値に前記メタデータを追加することによりオブジェクトとして定義されたデータを生成して前記出力コード内にオブジェクトとして定義されたデータを生成させる、変換するステップと、

前記処理ユニットが前記出力コードを解析し、前記値型として定義されたデータが前記オブジェクトとして定義されたデータを受け取ることを予想する関数に引き渡されると判断するステップと、

30

前記処理ユニットにより、前記値型として定義されたデータに代えて、前記オブジェクトとして定義されたデータを前記関数に引き渡すステップと

を含むことを特徴とする方法。

【請求項 11】

前記オブジェクトはユーザにより定義されたデータ型であり、前記メタデータは、前記データ型の型名、前記データ型に含まれるフィールドのフィールド名、前記フィールドのフィールドタイプ、および前記データ型に関して実行できるオペレーションを含むことを特徴とする請求項 10 に記載の方法。

40

【請求項 12】

前記メタデータは、前記オブジェクトに含まれた値がどのように解釈されるかを記述することを特徴とする請求項 10 または 11 に記載の方法。

【発明の詳細な説明】

【技術分野】

【0001】

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。

【背景技術】

50

【 0 0 0 2 】

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。

10

【 0 0 0 3 】

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。

20

【 0 0 0 4 】

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。

30

【 0 0 0 5 】

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。

40

【 0 0 0 6 】

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

50

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」を作ることができるにしていた。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答えていない。

【発明の概要】

【発明が解決しようとする課題】

【0009】

本発明は、なにかんずく、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。

【課題を解決するための手段】

【0010】

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、なにか、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。

10

【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。

【図面の簡単な説明】

【0018】

【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。

20

【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。

30

【発明を実施するための形態】

【0019】

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。

40

【0020】

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

50

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。

【0021】

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。

【0022】

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。

【0023】

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。

【0024】

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

10

20

30

40

50

Rect RectObject;

【 0 0 2 5 】

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。

【 0 0 2 6 】

組み込みデータ型 1 1 4 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） 1 1 8 を、コンパイラ 1 0 4 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则认为られている値型のことである。

【 0 0 2 7 】

同様に、基本オブジェクトクラス階層 1 1 6 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。

【 0 0 2 8 】

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タ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。

【 0 0 2 9 】

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

【 0 0 3 0 】

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 1 1 0 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 1 1 2 によってソースコードに表すことが可能である。

【 0 0 3 1 】

本発明の実施形態では、コンパイラ 1 0 4 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 1 1 8 を備えることが可能である。例えば、C言語で

10

20

30

40

50

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。

【0032】

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、開発者から見えなくコンパイルされ、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）。

【0033】

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。

【0034】

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。

【0035】

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

10

20

30

40

50

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。

【0036】

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。

【0037】

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。

【0038】

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クスやプログラマブルコンシューマエレクトロニクス、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り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。

【0039】

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、ROM 24 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。

【0040】

複数のプログラムモジュールをドライブと RAM 25 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 102、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。モニター 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。モニターのほかに、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。

【0041】

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。

【0042】

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 LAN 51 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、WAN 52 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。

【0043】

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ー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ー、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。

【0044】

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。

【0045】

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。

【0046】

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。

【0047】

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。

【0048】

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と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 3 5 4 はボックス化値型である。従って、組み込み値型 3 5 2 とユーザ定義の値型 3 5 4 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。

【 0 0 4 9 】

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 3 3 4）は、親のボックス化値型（例えば、point 値型 3 3 0）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は、ボックス化表現が期待されている場合にオブジェクトメソッドに渡すことができ、その逆も同じである。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。

【 0 0 5 0 】

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。

【 0 0 5 1 】

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。

【 0 0 5 2 】

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。

【 0 0 5 3 】

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

10

20

30

40

50

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。

【0054】

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。

【0055】

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。

【0056】

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。

【0057】

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。

【0058】

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

10

20

30

40

50

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。

【0059】

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、さもないければ、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。

【0060】

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、スタックウォーカー（stack walker）532、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）534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536が用意されていることもある。ランタイム環境は、ガーベッジコレクション（garbage collection）538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。

【0061】

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。

【0062】

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。

【0063】

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。

【 0 0 6 4 】

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。

【 0 0 6 5 】

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 6 0 0 からスタートし、アンボックス化テストオペレーション 6 0 2 に進む。テストオペレーション 6 0 2 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 6 0 2 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 6 0 4 に移る。アンボックス化値型出力オペレーション 6 0 4 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 6 0 8 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 6 1 9 に進み、変換された値型に対して終了する。

【 0 0 6 6 】

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。

【 0 0 6 7 】

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ると、アンボックス化テストオペレーション 6 0 2 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 6 1 0 に移ることである。ボックス化テストオペレーション 6 1 0 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。アンボックス化から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 6 1 2 に移り、変換された値型に対して終了する。

【 0 0 6 8 】

他方、アンボックス化から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 6 1 0 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。

【0069】

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。

【0070】

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。

【0071】

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。

【0072】

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。「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8に示すものがある。

【0073】

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。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。

【0074】

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

10

20

30

40

50

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。

【 0 0 7 5 】

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 7 0 4 に続いて、判断オペレーション 7 0 6 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。

10

【 0 0 7 6 】

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 7 0 6 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 7 0 8 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 7 0 8 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 7 0 8 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 7 0 8 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 7 0 8 に続いて、フローはコールオペレーション 7 1 0 へ進む。このオペレーションは以下で説明する。

20

【 0 0 7 7 】

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 7 0 6 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 7 1 0 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 7 0 8 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 7 1 0 を続ける。

【 0 0 7 8 】

コールオペレーション 7 1 0 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 - 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ス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。その代わりに、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。

30

【 0 0 7 9 】

従って、メソッドが 7 1 0 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 7 1 2 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ったら、実行オペレーション 7 1 4 はその値型に対するメソッドを実行する。

40

【 0 0 8 0 】

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

50

け取るかを知らないなので、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。

【 0 0 8 1 】

図 8 に代わる代替実施形態のオペレーションフロー 8 0 0 は図 9 に示されている。フロー 8 0 0 の最初の 2 オペレーションである、定義オペレーション 8 0 2 とパスオペレーション 8 0 4 は、図 8 を参照して説明したオペレーション 7 0 2 および 7 0 4 に類似している。つまり、定義オペレーション 8 0 2 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 8 0 4 はその値を関数に渡している。

【 0 0 8 2 】

値が渡されると、コールオペレーション 8 0 6 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 7 1 0 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 8 0 6 に続いて、ルックアップオペレーション 8 0 8 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 8 0 8 は、上述したオペレーション 7 1 2 に類似している。

【 0 0 8 3 】

コールオペレーション 8 0 6 とルックアップオペレーション 8 0 8 に続いて、判断オペレーション 8 1 0 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 8 1 0 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 7 0 6 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。

【 0 0 8 4 】

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 8 1 0 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 8 1 2 に移る。変更オペレーション 8 1 2 は、図 8 を参照して上述した変更オペレーション 7 0 8 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 8 1 2 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 8 1 4 は、変更された値型を使用してメソッドを実行する。

【 0 0 8 5 】

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 8 1 0 で判断されると、実行オペレーション 8 1 4 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 8 1 2 のような変更オペレーションは、実行前に必要でない。

【 0 0 8 6 】

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。コーラの数が増え、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。

【 0 0 8 7 】

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

10

20

30

40

50

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。

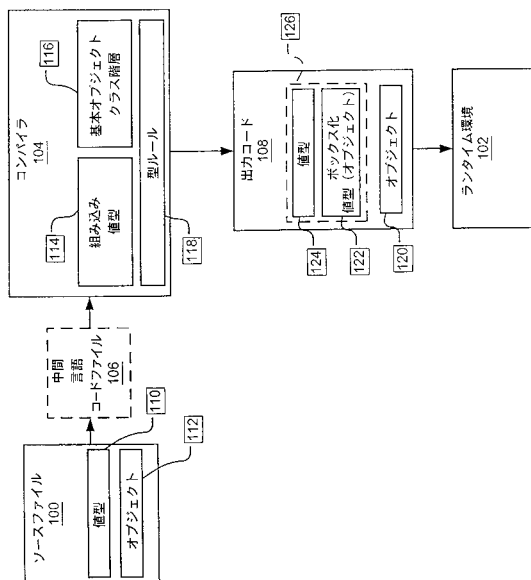
【符号の説明】

【0088】

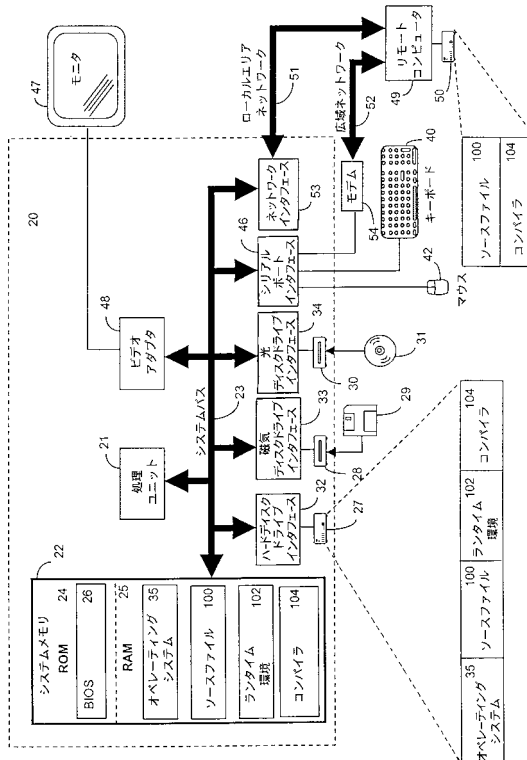
- 100 ソースファイル
- 102 ランタイム環境
- 104 コンパイラ
- 106 中間言語コードファイル
- 108 出力コード
- 110 値型
- 112 オブジェクト
- 114 組み込み値型
- 116 基本オブジェクトクラス階層
- 118 型ルール
- 120 基本オブジェクトクラス階層
- 122 型ルール
- 124 オブジェクト
- 126 ランタイム環境
- 302 組み込み型
- 304 ユーザ定義の型

10

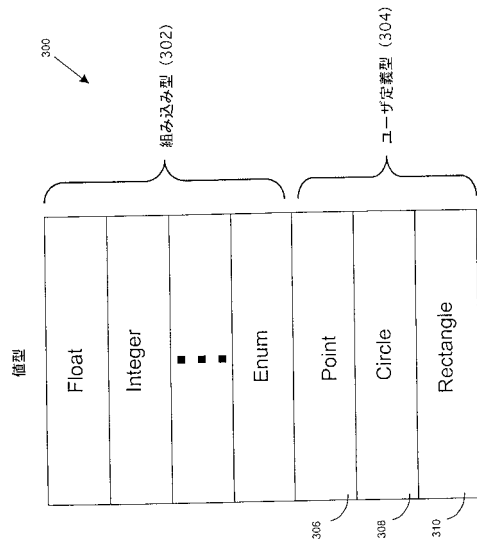
【図1】



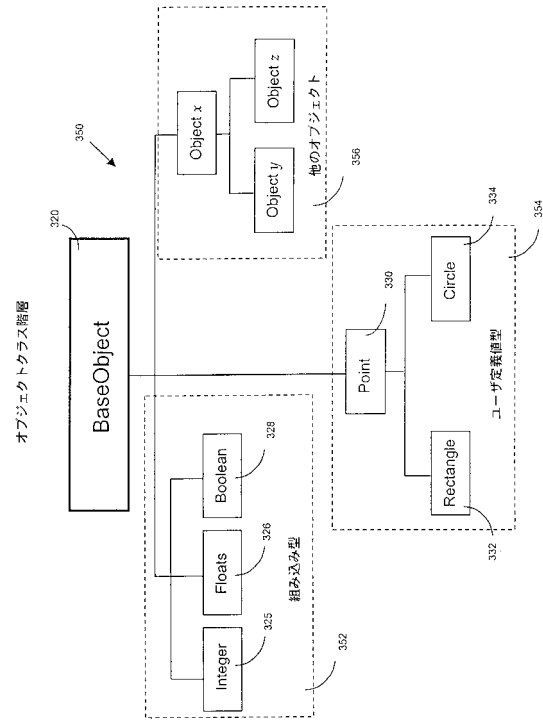
【図2】



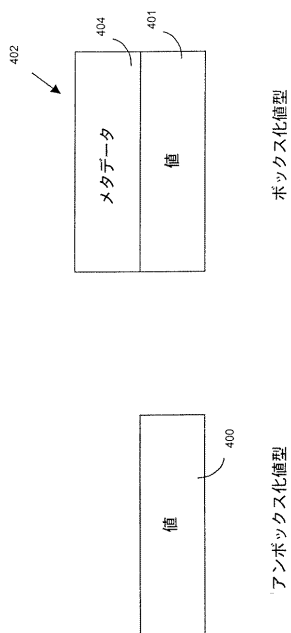
【図 3】



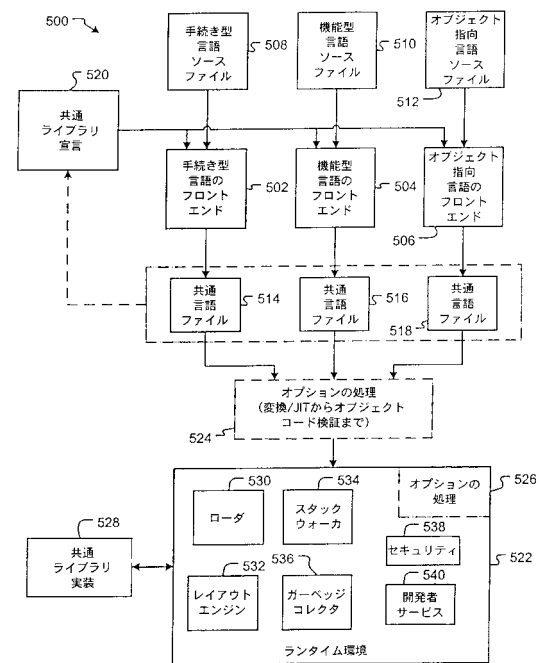
【図 4】



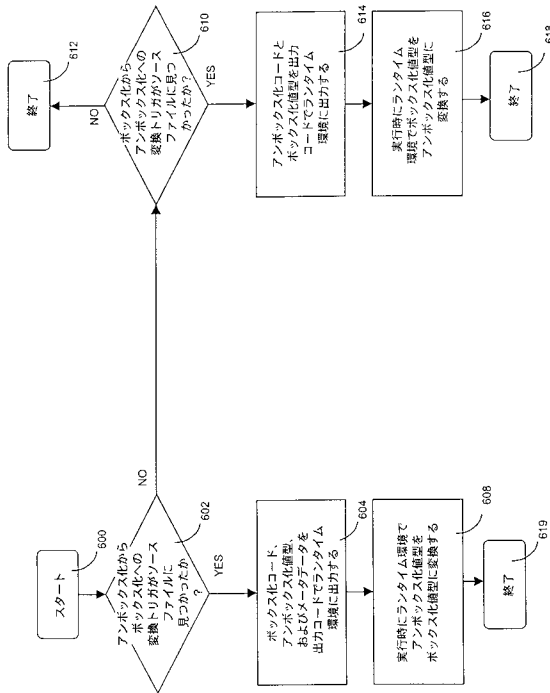
【図 5】



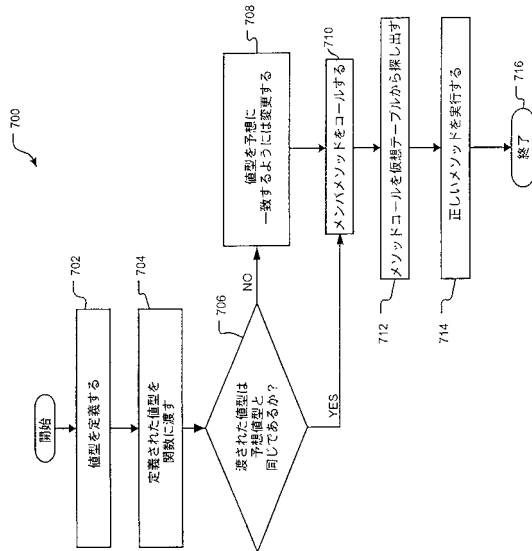
【図 6】



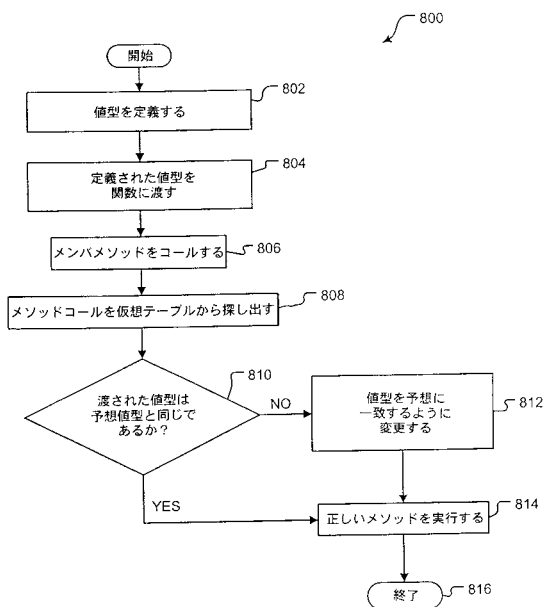
【図 7】



【図 8】



【図 9】



フロントページの続き

(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 17
213

(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

審査官 林 毅

(56)参考文献 特開平11-338875(JP,A)
特開平3-102427(JP,A)
高木 聡, C MAGAZINE , 日本, 株式会社日本ソフトバンク, 1990年 1月10
日, 第2巻、第1号, 第112頁
神門 有史、外2名, 関数型言語処理系におけるデータ構成子のunbox化, コンピュータソ
フトウェア, 日本, 日本ソフトウェア科学会, 1997年 7月15日, 第14巻、第4号, 第70
-75頁

(58)調査した分野(Int.Cl., DB名)
G06F 9/44
G06F 9/45