US 20140282441A1

(19) United States

(12) Patent Application Publication

(10) Pub. No.: US 2014/0282441 A1

(43) Pub. Date: Sep. 18, 2014

Hoban et al.

(54) **STATIC TYPE CHECKING ACROSS MODULE UNIVERSES**

(71) Applicant: **MICROSOFT CORPORATION**, Redmond, WA (US)

(72) Inventors: **Lucas J. Hoban**, Seattle, WA (US); **Mads Torgersen**, Issaquah, WA (US); **Charles P. Jazdzewski**, Redmond, WA (US); **Anders Hejlsberg**, Seattle, WA (US); **Steven E. Lucco**, Bellevue, WA (US); **Joseph J. Pamer**, Seattle, WA (US)

(73) Assignee: **Microsoft Corporation**, Redmond, WA (US)

(21) Appl. No.: **13/798,088**

(22) Filed: **Mar. 13, 2013**

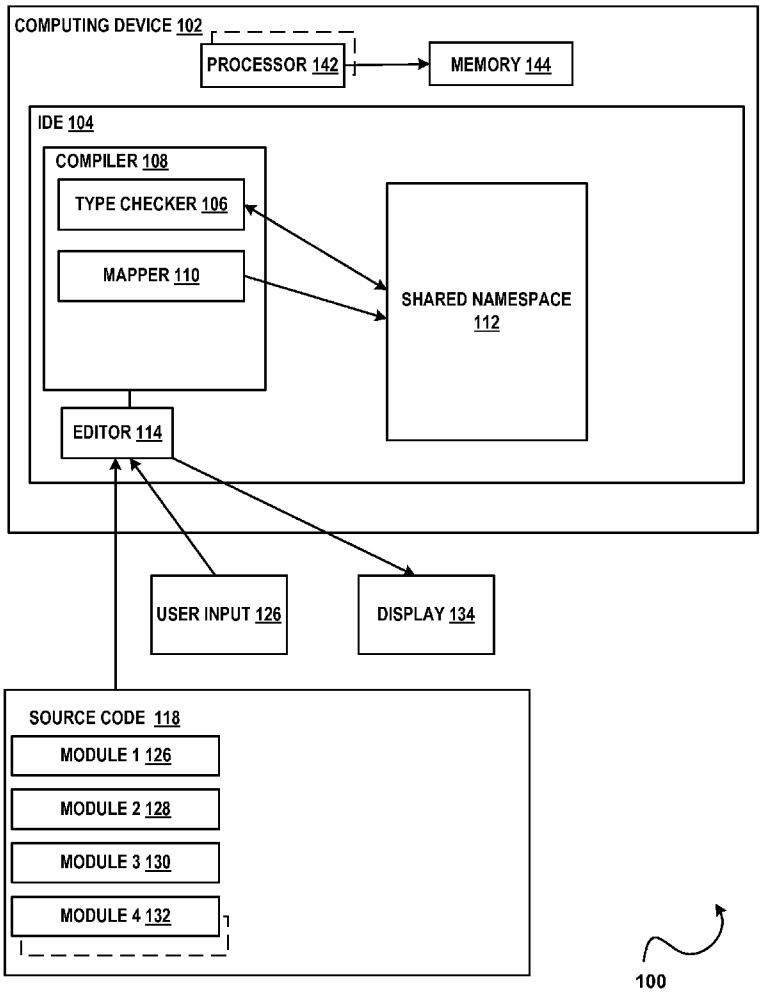
Publication Classification

(51) **Int. Cl.**
G06F 9/45 (2006.01)

(52) **U.S. Cl.**
CPC *G06F 8/41* (2013.01)
USPC **717/141**

(57) **ABSTRACT**

Static type checking can be performed on types and values defined in modules in a system that dynamically composes programs from modules. The types and values do not share a global namespace. Each module defines its own module universe, disjoint from other modules. A language mechanism can establish a local name binding to one module within the content of another module. When type checking at compile time an environment can be established that corresponds to a runtime instance of the program. The static type system can be arranged to align with the runtime values, such that the names used to refer to objects at runtime are the same as the names used to refer to the types of those objects in the static type system. Aliases of a particular type are resolved to a known compile time description of the type.



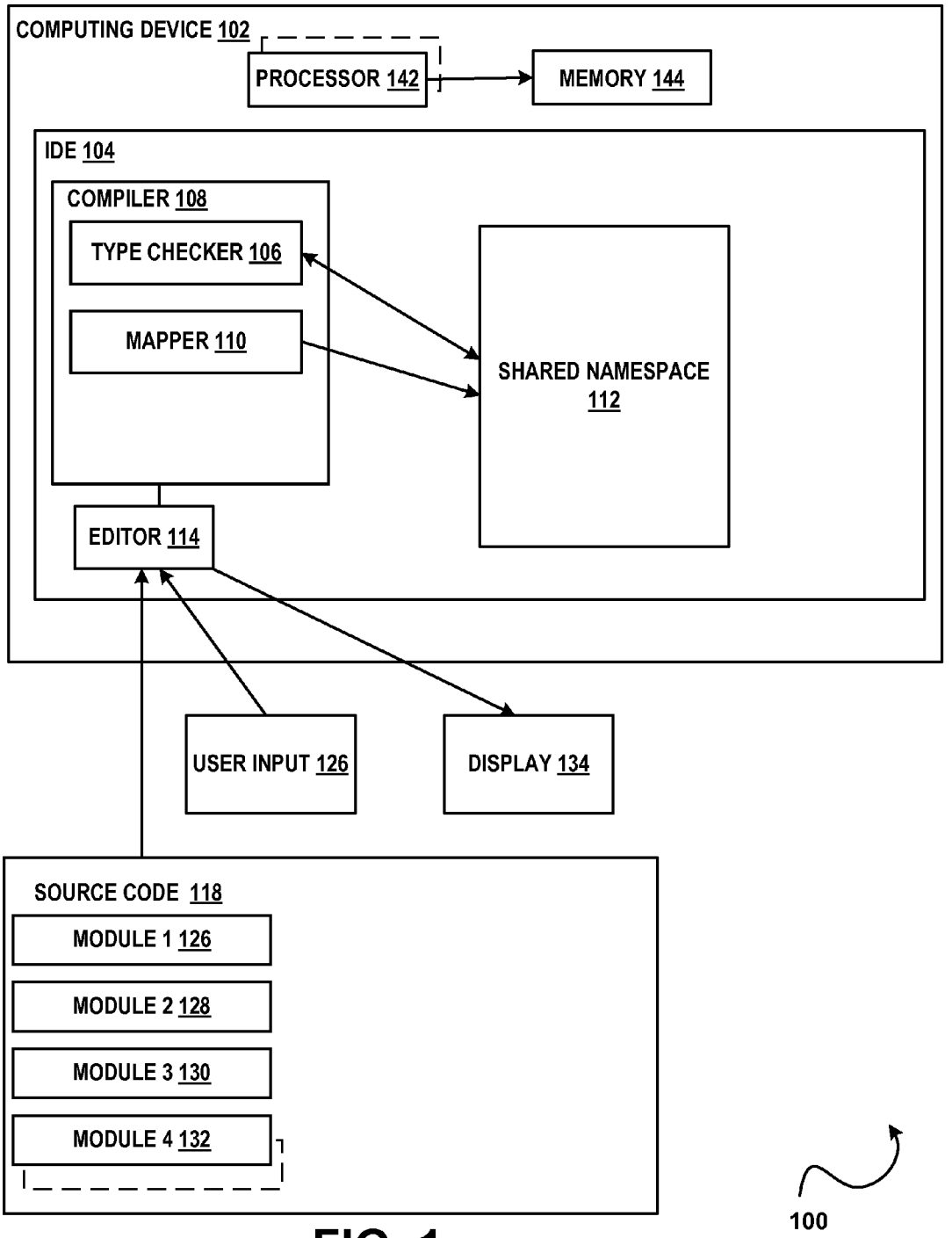

COMPUTING DEVICE 102
PROCESSOR 142
MEMORY 144
IDE 104
COMPILER 108
TYPE CHECKER 106
MAPPER 110
SHARED NAMESPACE 112
EDITOR 114
USER INPUT 126
DISPLAY 134
SOURCE CODE 118
MODULE 1 126
MODULE 2 128
MODULE 3 130
MODULE 4 132
100

FIG. 1a

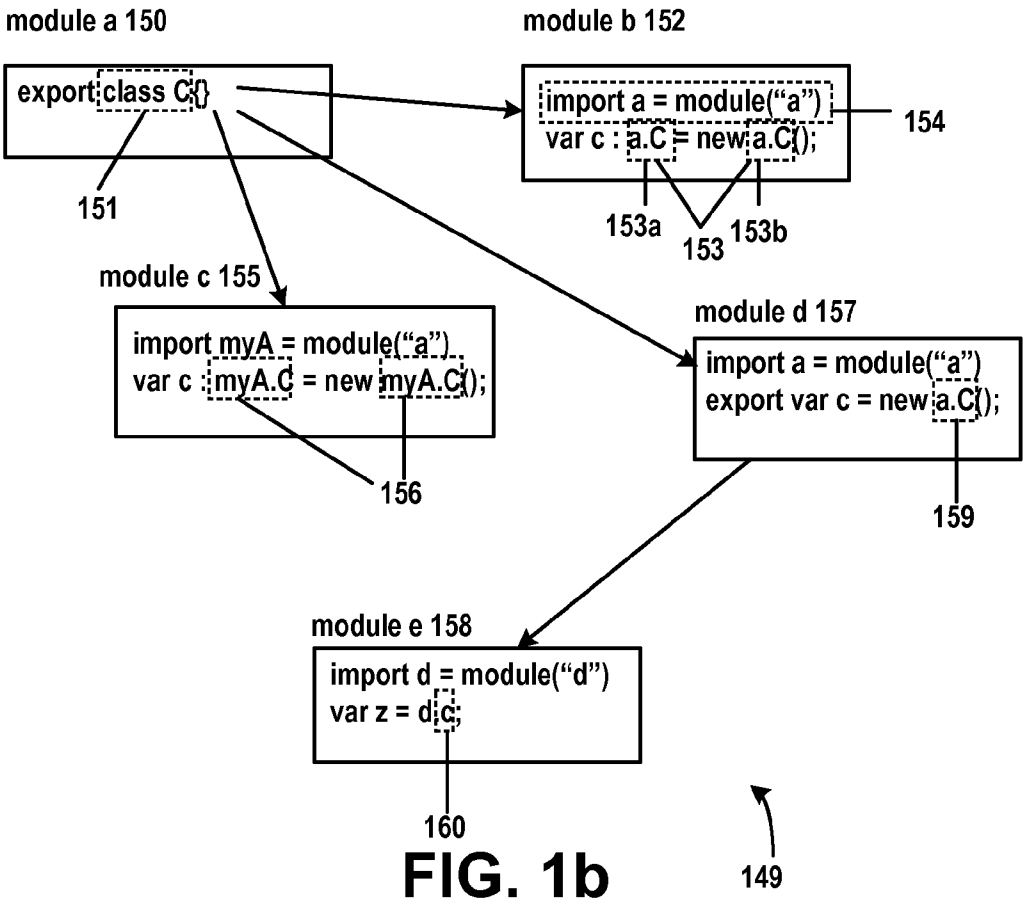
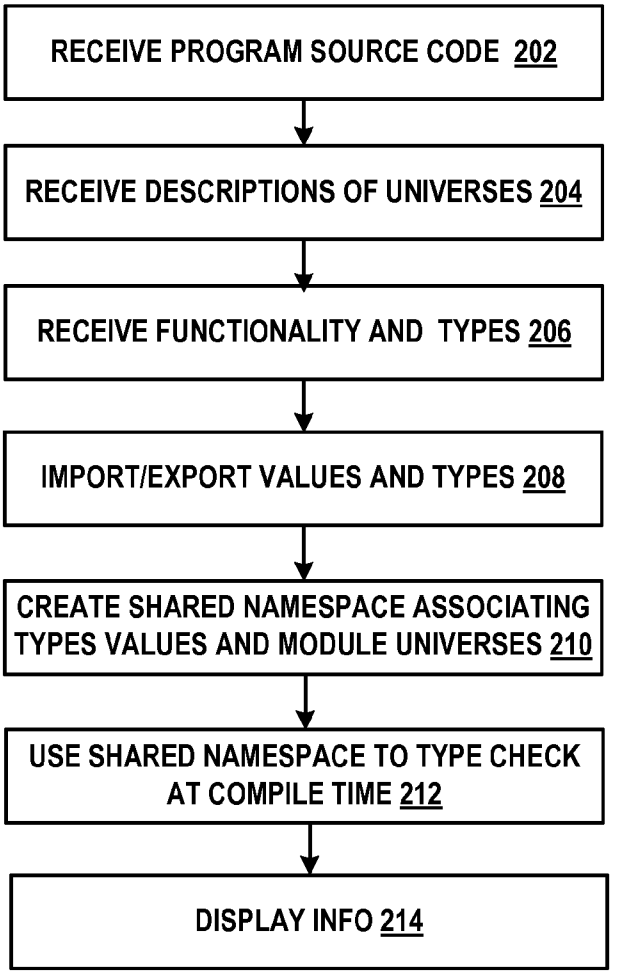


FIG. 1b

161

class C	module a 150	C 151
	module b 152	a.C 152
	module c 155	myA.C 156
	module d 157	a.C 159
	module e 158	<not denotable>

FIG. 1c


RECEIVE PROGRAM SOURCE CODE 202
RECEIVE DESCRIPTIONS OF UNIVERSES 204
RECEIVE FUNCTIONALITY AND TYPES 206
IMPORT/EXPORT VALUES AND TYPES 208
CREATE SHARED NAMESPACE ASSOCIATING TYPES VALUES AND MODULE UNIVERSES 210
USE SHARED NAMESPACE TO TYPE CHECK AT COMPILE TIME 212
DISPLAY INFO 214

200

FIG. 2

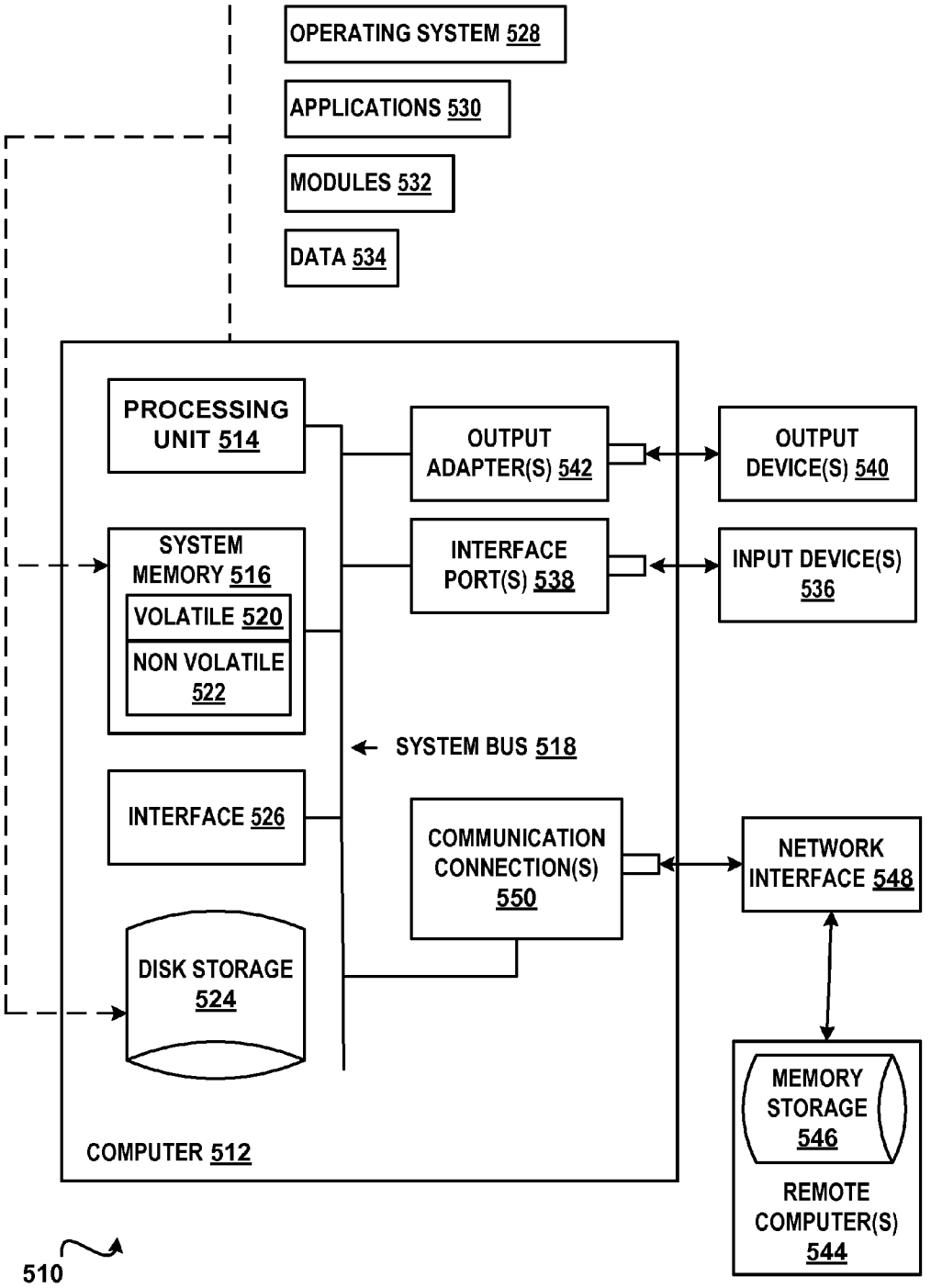

OPERATING SYSTEM 528
APPLICATIONS 530
MODULES 532
DATA 534
PROCESSING UNIT 514
SYSTEM MEMORY 516
VOLATILE 520
NON VOLATILE 522
INTERFACE 526
DISK STORAGE 524
SYSTEM BUS 518
OUTPUT ADAPTER(S) 542
INTERFACE PORT(S) 538
COMMUNICATION CONNECTION(S) 550
OUTPUT DEVICE(S) 540
INPUT DEVICE(S) 536
NETWORK INTERFACE 548
MEMORY STORAGE 546
REMOTE COMPUTER(S) 544
COMPUTER 512
510

FIG. 3

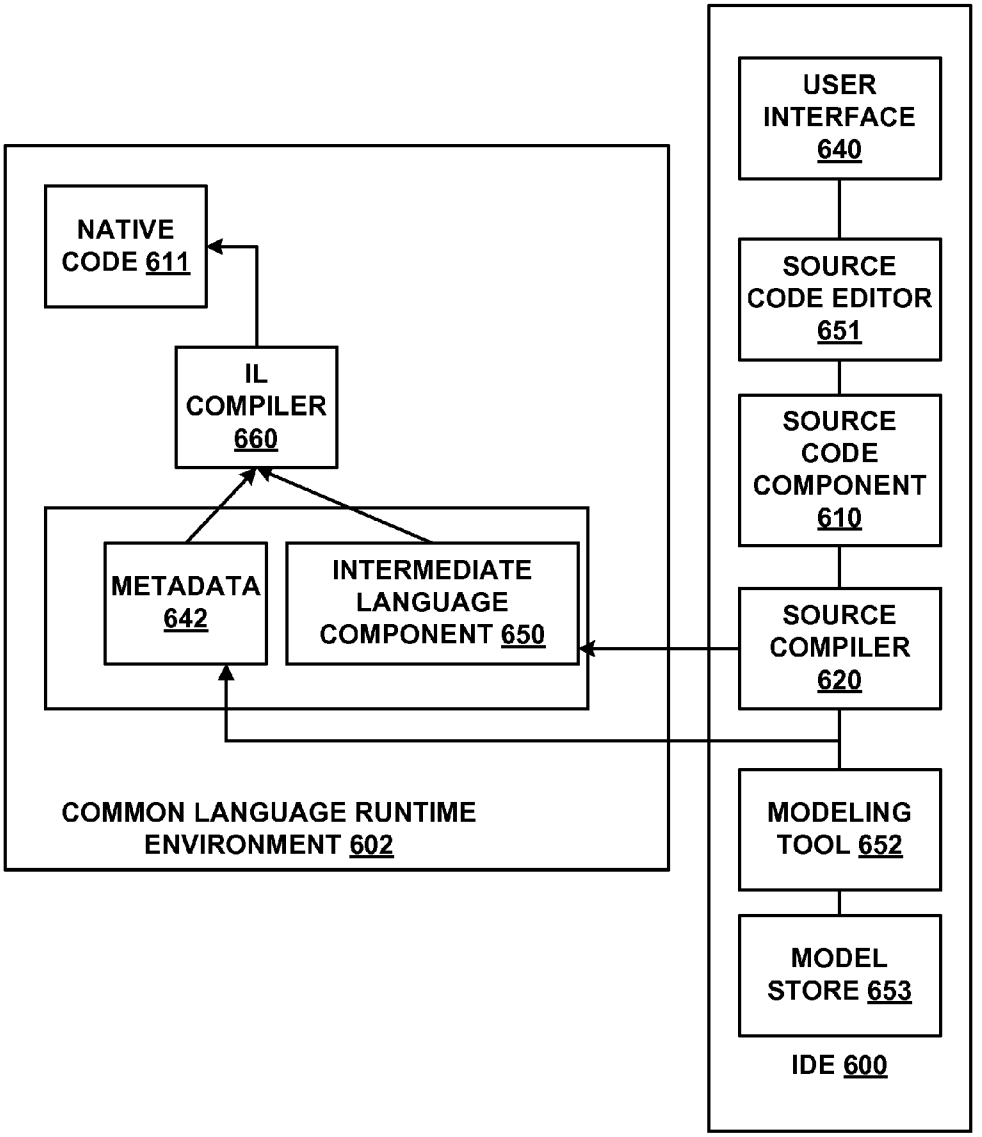

USER INTERFACE 640
SOURCE CODE EDITOR 651
SOURCE CODE COMPONENT 610
SOURCE COMPILER 620
MODELING TOOL 652
MODEL STORE 653
IDE 600
NATIVE CODE 611
IL COMPILER 660
METADATA 642
INTERMEDIATE LANGUAGE COMPONENT 650
COMMON LANGUAGE RUNTIME ENVIRONMENT 602

FIG. 4

STATIC TYPE CHECKING ACROSS MODULE UNIVERSES

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] The application is related in subject matter to co-pending U.S. patent application Ser. No. _____ (Docket No. 337890.01) entitled "STATICALLY EXTENSIBLE TYPES", filed on _____. The application is related in subject matter to co-pending U.S. patent application Ser. No. _____ (Docket No. 337891.01) entitled "CONTEXTUAL TYPING", filed on _____. The application is related in subject matter to co-pending U.S. patent application Ser. No. _____ (Docket No. 338454.01) entitled "GRAPH-BASED MODEL FOR TYPE SYSTEMS", filed on _____.

BACKGROUND

[0002] Computer programming languages support various data types that can be assigned to variables or expressions. Data types include strings, floating point numbers, integers and so on. Assigning the wrong data type to a variable or expression can cause a program to malfunction or to stop running. Type checking is one way to ensure that the wrong data type is not assigned to a variable or expression. Type checking can be performed at compile time (static type checking) or at runtime (dynamic type checking). Static type checking is able to verify that the type checked conditions hold for all possible executions of the program. Dynamic type checking is able to verify that a particular execution of a program is free of type errors. Therefore, dynamic type checking is typically performed each time the program is run.

SUMMARY

[0003] Static type checking can be performed on types and values defined in modules in a system that dynamically composes executable programs from modules. The term "module" as used herein refers to a separately loaded body of code that creates and initializes a singleton module instance. A module can be referenced using an external module name. As used herein, modules are part of the statically typed programming language, and the statements and declarations express concepts from the statically typed programming language. In dynamically composing systems, each module exists in a separate namespace, such that variable declarations do not conflict or interact across modules. However, explicit declarative imports may be used within a module to give locally scoped names to entities defined in other modules. A language mechanism can be used to establish a local name binding to one module within the content of another module. A module data type is provided by a language concept in which a module is a container for other data types. Within a module, declarations of various variables, classes, other modules, etc. can exist. Data hiding or encapsulation can be implemented in a language that does not support data encapsulation by having different visibility constraints placed on the variables, classes, other modules, etc. declared within each module.

[0004] A "visible" constraint enables the variables, classes, other modules, etc. to be visible wherever the module is defined. A second ("not visible") constraint allows the variable, class, etc. to be visible only in the module or file in which the module is declared. The static type system can determine dependencies of modules at compile time based on declarative imports. The static type system can map names of types and values defined in a plurality of modules in a way that is consistent with dynamic name binding of objects. When type checking at compile time, an environment can be established that corresponds to a runtime instance of the program. A language mechanism can be used to perform mapping between a static type and a runtime value. Aliases of a particular type are resolved to a known compile time description of the type.

[0005] This Summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used to limit the scope of the claimed subject matter.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] In the drawings:

[0007] FIG. **1***a* illustrates an example of a system **100** that enables static type checking across module universes in accordance with aspects of the subject matter described herein;

[0008] FIG. **1***b* illustrates an example 149 of interdependencies between modules in accordance with aspects of the subject matter disclosed herein;

[0009] FIG. **1***c* illustrates an example of a table **161** that represents mapping between various names representing the same thing in accordance with aspects of the subject matter disclosed herein;

[0010] FIG. **2** illustrates an example of a method **200** that performs static type checking across module universes in accordance with aspects of the subject matter disclosed herein;

[0011] FIG. **3** is a block diagram of an example of a computing environment in accordance with aspects of the subject matter disclosed herein.

[0012] FIG. **4** is a block diagram of an example of an integrated development environment (IDE) in accordance with aspects of the subject matter disclosed herein.

DETAILED DESCRIPTION

Overview

[0013] A static type system typically establishes a global namespace of values and types, and associates each expression and statement in the program with data types that can be validly assigned to the expression based on the global namespace of values and types. At compile time, the static type checker checks that the data types associated with the expressions in the source code are correct.

[0014] A dynamic runtime system that uses modules for composing programs can segment the runtime view of an executing program into separate disjoint module universes. Running programs in disjoint module universes can be useful for isolation of execution because different modules are compiled and executed separately. However, static type checking of the aggregate program is challenging because the full system of modules which will be loaded dynamically has to be known in advance and relationships between modules have to be established before execution. In dynamic systems, dependencies are resolved as modules are loaded. It is difficult for a static system to understand interdependencies between modules statically. (Statically understanding inter-

dependencies that exist between module universes at runtime is challenging for a static type system executing at compile time.)

[0015] Disjoint module universes can refer to instances of the same element using different names. It is difficult for a static system to know how to map elements referred to by different names to the correct type. Finally, types in a module universe of a dynamic runtime system can come from other module universes, which the module universe understands because of dynamically resolving dependencies. It is difficult for a static universe to assign locally scoped names to remotely defined types and instances. For at least these reasons, static type languages typically avoid the use of disjoint module universes. Similarly, dynamic languages that use a module system do not try to have a static view of the module system. Static type checking and therefore tooling based on static type systems is thus unavailable at development time for traditional dynamic module-based runtime systems.

[0016] In accordance with aspects of the subject matter described herein, a static type system accommodates the use of disjoint module universes at runtime by mapping types defined in disjoint module universes into other module universes in a way that is consistent with the dynamic name binding of object instances. The type definition associated with a module can be statically determined based on type checking. Type definition of the module can be based on the set of definitions in the module. In accordance with aspects of the subject matter described herein, the type system's view of modules can include both values and types. The type system's view of modules can include elements such as but not limited to classes that include both values and types. Static type and the runtime value can be mapped to each other's local name binding. Aliases of a type can be understood to refer to the same type.

[0017] In accordance with some aspects of the subject matter described herein, an example of a programming language supporting static type checking across module universes in a dynamic runtime is TypeScript. Typescript is a superset of JavaScript, meaning that any valid JavaScript code is valid TypeScript code. TypeScript adds additional features on top of JavaScript. TypeScript can be converted into JavaScript code by the TypeScript compiler. TypeScript permits the annotation of variables, function arguments, and functions with type information, facilitating the use of static typing to provide tools such as auto-completion tools and enabling more comprehensive error checking than that provided by using traditional JavaScript.

[0018] In accordance with some aspects of the subject matter described herein, declarations introduce names in the declaration spaces to which they belong. The global module has a declaration space for global members (variables, functions, modules, and constructor functions), and a declaration space for global types (modules, classes, and interfaces). Each module has a declaration space for local members (variables, functions, modules, and constructor functions), and a declaration space for local types (modules, classes, and interfaces). Every declaration (whether local or exported) in a module contributes to one or both of these declaration spaces. Each module has a declaration space for exported members and a declaration space for exported types. All export declarations in the module contribute to these declaration spaces.

[0019] Top-level declarations in a non-module source file belong to the global module. Top-level declarations in a module source file belong to the external module represented by that source file.

[0020] An export declaration can declare an externally accessible module member including but not limited to members such as, for example, an export variable statement, function declaration, class declaration, interface declaration or module declaration. An export declaration can be distinguished from a regular declaration by, for example, being prefixed with a keyword such as "export", etc. Exported variable, function, class, and module declarations can become properties on the module instance and together can establish the module's instance type. The module's instance type can have any combination of the following members: a property for each exported variable declaration, a property of a function type for each exported function declaration, a property of a constructor type for each exported class declaration and/or a property of an object type for each exported internal module declaration.

[0021] In accordance with aspects of the subject matter described herein, module instance types can be referenced by using module identifiers as type names. An exported member can depend on a set of named types. The named types upon which a member depends can be the named types occurring in the transitive closure of the "directly depends on" relationship defined as follows:

[0022] A variable directly depends on its type.

[0023] A function directly depends on its function type.

[0024] A class directly depends on its constructor function type and its class instance type.

[0025] An interface directly depends on the type it declares.

[0026] A module directly depends on its module instance type.

[0027] An object type directly depends on the types of each of its public properties and the parameter and return types of each of its call, construct, and index signatures.

[0028] The term "module" as used herein refers to a body of statements and declarations that create and initialize a singleton module instance. Members exported from a module become properties on the module instance. The body of a module corresponds to a function that is executed once, thereby providing a mechanism for isolating local state. In accordance with aspects of the subject matter described herein, TypeScript supports modules. A module can be a separately loaded body of code referenced using an external module name. A module can be written as a separate source file that includes at least one import or export declaration.

[0029] An export declaration can be identified by an "export" prefix or by any other suitable means. Exported variable, function, class, and module declarations can become properties on the module instance and together establish the module's instance type. The exported type can have the following members:

[0030] A property for each exported variable declaration.

[0031] A property of a function type for each exported function declaration.

[0032] A property of a constructor type for each exported class declaration.

[0033] An exported member depends on a set of named types. The set of named types can be the empty set. The named types upon which a member depends can be the named

types occurring in the transitive closure of the types of all members of named types visible from a given location as follows:

[0034] A variable directly depends on its type.

[0035] A function directly depends on its function type.

[0036] A class directly depends on its constructor function type and its class instance type.

[0037] An interface directly depends on the type it declares.

[0038] An object type directly depends on the types of each of its public properties and the parameter and return types of each of its call, construct, and index signatures.

[0039] Import declarations can be used to import modules and to create local aliases by which the modules may be referenced. An import declaration can introduce a local identifier that references a given module. The local identifier can itself be classified as a module and can behave like a module. A string literal specified in an module reference can be interpreted as an module name. Import declarations in modules can specify either external module references or module names.

[0040] Modules can be separately loaded bodies of code referenced using external module names. Modules can be written as separate source files that contain at least one import declaration or export declaration. Below is an example of two modules written in separate source files.

```
File main.ts:
import log = module("log");
log.message("hello");
File log.ts:
export function message(s: string) {
    console.log(s);
}
```

[0041] In this example, two files define the "main" and "log" modules. The "main" module imports the "log" module and then uses functionality ("message") exported from the "log" module, passing it a value "hello". The "log" module exports a function that provides a way to log onto the console.

[0042] In the file "main.ts", the import declaration references the "log" module. Compiling the "main.ts" file causes the "log.ts" file to also be compiled as part of the program. At runtime, the import declaration loads the "log" module and produces a reference to the "log" module instance through which it is possible to reference the function exported by the "log" module.

[0043] Modules can be identified and referenced using external module names. The set of legal module names is defined by the runtime module loader system that the compiler for the statically typed language targets.

Static Type Checking Across Module Universes

[0044] FIG. **1***a* illustrates a block diagram of an example of a system **100** in accordance with aspects of the subject matter described herein. All or portions of system **100** may reside on one or more computers or computing devices such as the computers described below with respect to FIG. **3**. System **100** or portions thereof may be provided as a stand-alone system or as a plug-in or add-in. System **100** or portions thereof may include information obtained from a service (e.g., in the cloud) or may operate in a cloud computing environment. A cloud computing environment can be an environment in which computing services are not owned but are provided on demand. For example, information may reside on multiple devices in a networked cloud and/or data can be stored on multiple devices within the cloud. System **100** may execute in whole or in part on a software development computer such as the software development computer described with respect to FIG. **4**. All or portions of system **100** may be operated upon by program development tools. For example, all or portions of system **100** may execute within an integrated development environment (IDE) such as for example IDE 104. IDE 104 may be an IDE as described more fully with respect to FIG. **4** or can be another IDE. System **100** can execute wholly or partially outside an IDE.

[0045] System **100** can include one or more computing devices such as, for example, computing device **102**. A computing device such as computing device **102** can include one or more processors such as processor **142**, etc., and a memory such as memory **144** connected to the one or more processors. Computing device **102** can include one or more components comprising a compiler such as compiler **108**. A compiler such as compiler **108** may be a computer program or set of programs that translates text written in a (typically high-level) programming language into another (typically lower-level) computer language (the target language). The output of the compiler may be object code. Typically the output is in a form suitable for processing by other programs (e.g., a linker), but the output may be a human-readable text file. Source code is typically compiled to create an executable program but may be processed by program development tools which may include tools such as editors, beautifiers, static analysis tools, refactoring tools and others that operate in background or foreground.

[0046] A compiler **108** may comprise a .NET compiler that compiles source code written in a .NET language to intermediate byte code. .NET languages include but are not limited to C#, C++, F#, J#, JScript.NET, Managed Jscript, IronPython, IronRuby, VBx, VB.NET, Windows PowerShell, A#, Boo, Cobra, Chrome (Object Pascal for .NET, not the Google browser), Component Pascal, IKVM.NET, IronLisp, L#, Lexico, Mondrian, Nemerle, P#, Phalanger, Phrogram, PowerBuilder, #Smalltalk, AVR.NET, Active Oberon, APLNext, Common Larceny, Delphi.NET, Delta Forth .NET, DotLisp, EiffelEnvision, Fortran .NET, Gardens Point Modula-2/CLR, Haskell for .NET, Haskell.net, Hugs for .NET, IronScheme, LOLCode.NET, Mercury on .NET, Net Express, NetCOBOL, OxygenScheme, S#, sml.net, Wildcat Cobol, X# or any other .NET language. Compiler **108** may comprise a JAVA compiler that compiles source code written in JAVA to byte code. Compiler **108** can be any compiler for any programming language including but not limited to Ada, ALGOL, SMALL Machine Algol Like Language, Ateji PX, BASIC, BCPL, C, C++, CLIPPER 5.3, C#, CLEO, CLush, COBOL, Cobra, Common Lisp, Corn, Curl, D, DASL, Delphi, DIBOL, Dylan, dylan.NET, eC (Ecere C), Eiffel, Sather, Ubercode, eLisp Emacs Lisp, Erlang, Factor, Fancy, Formula One, Forth, Fortran, Go, Groovy, Haskell, Harbour, Java, JOVIAL, LabVIEW, Nemerle, Obix, Objective-C, Pascal, Plus, ppC++, RPG, Scheme, Smalltalk, ML, Standard ML, Alice, OCaml, Turing, Urq, Vala, Visual Basic, Visual FoxPro, Visual Prolog, WinDev, X++, XL, and/or Z++. Compiler **108** can be a compiler for any typed programming language.

[0047] A compiler such as compiler **108** and/or program development tools are likely to perform at least some of the following operations: preprocessing, lexical analysis, parsing (syntax analysis), semantic analysis, code generation, and

code optimization. Compiler **108** may include one or more modules comprising a parser that receives program source code and generates a parse tree.

[0048] System **100** can include one or more program components such as type checker **106** that performs static type checking across module universes as described herein. Type checker **106** can be a part of a compiler **108** as illustrated in FIG. **1***a*, or can be part of another program development tool (not shown). Type checker **106** can be a separate entity, plug-in, or add-on (not shown). It will be appreciated that program components such as for example, type checker **106** that performs static type checking across module universes can be loaded into memory **144** to cause one or more processors such as processor **142**, etc. to perform the actions attributed to type checker **106**. System **100** can include any combination of one or more of the following: an editor such as but not limited to editor **114**, a display device such as display device **134**, and so on. Editor **114** can receive source code such as source code **118** and user input such as user input **126**. Results such as the results of static type checking on dynamically composed module universes can be displayed on display device **134**. Other components well known in the arts may also be included but are not here shown.

[0049] A compiler such as compiler **108** can receive source code in the form of one or more program modules such as, for example, module 1 **126**, module 2 **128**, module 3 **130**, module 4, **132**, etc. The program modules may represent a TypeScript program. In accordance with aspects of the subject matter disclosed herein, a TypeScript program may include one or more source code files. A source file can be an implementation source file. A source file can be a declaration source file. In accordance with some aspects of the subject matter described herein, implementation source files and declaration source files can be distinguished by the file extension used. For example, source files with extension '.ts' can be implementation source files that include statements and declarations. Source files with extension '.d.ts' can be declaration source files that include declarations. Source code files can include export and import statements that are used to implement static type checking across module universes as described more fully below.

[0050] System **100** can include one or more compiler components such as mapper **110** that can create a shared namespace **112** used to perform static type checking across module universes as described herein. Mapper **110** can be a part of a compiler **108** as illustrated in FIG. **1***a*, or can be part of another program development tool (not shown). Mapper **110** can be a separate entity, plug-in, or add-on (not shown). It will be appreciated that compiler components such as for example, mapper **110** that performs static type checking across module universes can be loaded into memory **144** to cause one or more processors such as processor **142**, etc. to perform the actions attributed to mapper **110**.

[0051] FIG. **1***b* illustrates examples of modules and relationships between the modules The modules can be loaded as independent files. In FIG. **1***b*, module a **150** exports a class C **151**. Module b **152** imports module a **150** and can refer to class C **151** by any specified name. In this case, the name used to refer to class C is "a.C" **153**, because module a **150** was imported by module b **152** ("import a=module("a") **154** class A was given the local name "a" within module b **152**. When module b **152** is type checked at compile time, the compiler will create an environment for type checking in which class C is referred to by the name "a.C" **153** in module b **152**. (See FIG. **1***c*, table **161**). The syntax of the statement "import a=module("a")" **154** is understood by the mapper **110** to indicate that module b **152** has a dependency on module a **150**.

[0052] Module b **152** arbitrarily refers to elements imported from module a **150** as "a.[something]", in this case, Class C of module a **150** is referred to by module b **152** as "a.C" **153**. Module b **152** provides an example of arranging the nomenclature of the static type system to align with the nomenclature of the runtime system's values. That is, the name used to refer to the value of var c at runtime, i.e., the name "a.C" **153***b* is the same as the name used to refer to the type of var c in the static type system, "a.C" **153***a*. Module c **155** also imports module a **150** but refers to elements in module a **150** as "myA.[something]", e.g., "myA.C" **156**. Within the context of module c **155** class C **151** in module a **150** is referred to as "myA.C" whereas within the context of module b **152** class C **151** in module a **150** is referred to as a.C **153**. Thus the same element (class C) present in two modules are referred to by different names. Types can be exposed within a module which do not have a local name provided by the program text. This is illustrated in modules d **157** and module e **158**. Module d **157** imports module a **150** and exports variable c ("a.C" **159**) whose type is the type of the class C**151** of module a **150**.

[0053] Module e **158** imports module d **157** but does not import module a **150**, so the variable "var z" **160** has the type of class C **151** defined in module a **150**, but no local name is available in module e **158** to refer to class C. The type checker can allow type checking and type inference to proceed using a compiler-internal notion of the universe in which the name was defined. The compiler's internal representation of the universe in which the name was defined can be presented to developers by for example, using the file path to the module definition or other identifying characteristics to uniquely identify the module.

[0054] FIG. **1***c* illustrates a table representation **161** of the different names for class C used in different modules (shared namespace **112** of FIG. **1***a*). In Table **161** class C is represented in module a **150** as C **151**, in module b **152** as a.C **152**, in module c **155** as myA.C **156**, in module d **157** as a.C **159** and class C, as described above, is used, but is not denotable.

[0055] FIG. **2** illustrates an example of a method **200** for static type checking across module universes in accordance with aspects of the subject matter disclosed herein. The method described in FIG. **2** can be practiced by a system such as but not limited to the one described with respect to FIG. **1***a* and for which an example was provided in FIGS. **1***b*-**1***c*. While method **200** describes a series of operations that are performed in a sequence, it is to be understood that method **200** is not limited by the order of the sequence. For instance, some operations may occur in a different order than that described. In addition, one operation may occur concurrently with another operation. In some instances, not all operations described are performed.

[0056] At operation **202** inputs can be received by a tool or compiler for a compilation. The inputs can describe a set of module universes in program source code. At operation **204** the compiler can get a set of files or other type of assets that describe the module universes. At operation **206** each of the module universes can provide a set of runtime functionality and a set of types. At operation **208** the module universes can import a set of values and types from other module universes and/or can export a set of values and types to other module

universes. At operation **210** the compiler or tool can establish a shared space that is not a global namespace for types and values, where each type and value is associated with the module universe the type or value came from. At operation **212** static type checking can be performed on each module universe using the shared space. Data can be type checked using module universes distinguished by which module universe the type or value comes from. At operation **214** the results of static type checking on dynamically composed module universes can be displayed.

Example of a Suitable Computing Environment

[0057] In order to provide context for various aspects of the subject matter disclosed herein, FIG. **3** and the following discussion are intended to provide a brief general description of a suitable computing environment **510** in which various embodiments of the subject matter disclosed herein may be implemented. While the subject matter disclosed herein is described in the general context of computer-executable instructions, such as program modules, executed by one or more computers or other computing devices, those skilled in the art will recognize that portions of the subject matter disclosed herein can also be implemented in combination with other program modules and/or a combination of hardware and software. Generally, program modules include routines, programs, objects, physical artifacts, data structures, etc. that perform particular tasks or implement particular data types. Typically, the functionality of the program modules may be combined or distributed as desired in various embodiments. The computing environment **510** is only one example of a suitable operating environment and is not intended to limit the scope of use or functionality of the subject matter disclosed herein.

[0058] With reference to FIG. **3**, a computing device in the form of a computer **512** is described. Computer **512** may include at least one processing unit **514**, a system memory **516**, and a system bus **518**. The at least one processing unit **514** can execute instructions that are stored in a memory such as but not limited to system memory **516**. The processing unit **514** can be any of various available processors. For example, the processing unit **514** can be a graphics processing unit (GPU). The instructions can be instructions for implementing functionality carried out by one or more components or modules discussed above or instructions for implementing one or more of the methods described above. Dual microprocessors and other multiprocessor architectures also can be employed as the processing unit **514**. The computer **512** may be used in a system that supports rendering graphics on a display screen. In another example, at least a portion of the computing device can be used in a system that comprises a graphical processing unit. The system memory **516** may include volatile memory **520** and nonvolatile memory **522**. Nonvolatile memory **522** can include read only memory (ROM), programmable ROM (PROM), electrically programmable ROM (EPROM) or flash memory. Volatile memory **520** may include random access memory (RAM) which may act as external cache memory. The system bus **518** couples system physical artifacts including the system memory **516** to the processing unit **514**. The system bus **518** can be any of several types including a memory bus, memory controller, peripheral bus, external bus, or local bus and may use any variety of available bus architectures. Computer **512** may include a data store accessible by the processing unit **514** by way of the system bus **518**. The data store may include executable instructions, 3D models, materials, textures and so on for graphics rendering.

[0059] Computer **512** typically includes a variety of computer readable media such as volatile and nonvolatile media, removable and non-removable media. Computer readable media may be implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer readable media include computer-readable storage media (also referred to as computer storage media) and communications media. Computer storage media includes physical (tangible) media, such as but not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CDROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices that can store the desired data and which can be accessed by computer **512**. Communications media include media such as, but not limited to, communications signals, modulated carrier waves or any other intangible media which can be used to communicate the desired information and which can be accessed by computer **512**.

[0060] It will be appreciated that FIG. **3** describes software that can act as an intermediary between users and computer resources. This software may include an operating system **528** which can be stored on disk storage **524**, and which can allocate resources of the computer **512**. Disk storage **524** may be a hard disk drive connected to the system bus **518** through a non-removable memory interface such as interface **526**. System applications **530** take advantage of the management of resources by operating system **528** through program modules **532** and program data **534** stored either in system memory **516** or on disk storage **524**. It will be appreciated that computers can be implemented with various operating systems or combinations of operating systems.

[0061] A user can enter commands or information into the computer **512** through an input device(s) **536**. Input devices **536** include but are not limited to a pointing device such as a mouse, trackball, stylus, touch pad, keyboard, microphone, voice recognition and gesture recognition systems and the like. These and other input devices connect to the processing unit **514** through the system bus **518** via interface port(s) **538**. An interface port(s) **538** may represent a serial port, parallel port, universal serial bus (USB) and the like. Output devices (s) **540** may use the same type of ports as do the input devices. Output adapter **542** is provided to illustrate that there are some output devices **540** like monitors, speakers and printers that require particular adapters. Output adapters **542** include but are not limited to video and sound cards that provide a connection between the output device **540** and the system bus **518**. Other devices and/or systems or devices such as remote computer(s) **544** may provide both input and output capabilities.

[0062] Computer **512** can operate in a networked environment using logical connections to one or more remote computers, such as a remote computer(s) **544**. The remote computer **544** can be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the computer **512**, although only a memory storage device **546** has been illustrated in FIG. **3**. Remote computer(s) **544** can be logically connected via communication connection(s) **550**. Network interface **548** encompasses communication networks such as local area networks (LANs)

and wide area networks (WANs) but may also include other networks. Communication connection(s) **550** refers to the hardware/software employed to connect the network interface **548** to the bus **518**. Communication connection(s) **550** may be internal to or external to computer **512** and include internal and external technologies such as modems (telephone, cable, DSL and wireless) and ISDN adapters, Ethernet cards and so on.

[0063] It will be appreciated that the network connections shown are examples only and other means of establishing a communications link between the computers may be used. One of ordinary skill in the art can appreciate that a computer **512** or other client device can be deployed as part of a computer network. In this regard, the subject matter disclosed herein may pertain to any computer system having any number of memory or storage units, and any number of applications and processes occurring across any number of storage units or volumes. Aspects of the subject matter disclosed herein may apply to an environment with server computers and client computers deployed in a network environment, having remote or local storage. Aspects of the subject matter disclosed herein may also apply to a standalone computing device, having programming language functionality, interpretation and execution capabilities.

[0064] FIG. **4** illustrates an integrated development environment (IDE) **600** and Common Language Runtime Environment **602**. An IDE 600 may allow a user (e.g., developer, programmer, designer, coder, etc.) to design, code, compile, test, run, edit, debug or build a program, set of programs, web sites, web applications, and web services in a computer system. Software programs can include source code (component **610**), created in one or more source code languages (e.g., Visual Basic, Visual J#, C++. C#, J#, Java Script, APL, COBOL, Pascal, Eiffel, Haskell, ML, Oberon, Pert, Python, Scheme, Smalltalk and the like). The IDE 600 may provide a native code development environment or may provide a managed code development that runs on a virtual machine or may provide a combination thereof. The IDE 600 may provide a managed code development environment using the Microsoft .NET™ framework. An intermediate language component **650** may be created from the source code component **610** and the native code component **611** using a language specific source compiler **620** using a modeling tool **652** and model store **653** and the native code component **611** (e.g., machine executable instructions) is created from the intermediate language component **650** using the intermediate language compiler **660** (e.g. just-in-time (JIT) compiler), when the application is executed. That is, when an intermediate language (IL) application is executed, it is compiled while being executed into the appropriate machine language for the platform it is being executed on, thereby making code portable across several platforms. Alternatively, in other embodiments, programs may be compiled to native code machine language (not shown) appropriate for its intended platform.

[0065] A user can create and/or edit the source code component according to known software programming techniques and the specific logical and syntactical rules associated with a particular source language via a user interface **640** and a source code editor **651** in the IDE 600. Thereafter, the source code component **610** can be compiled via a source compiler **620**, whereby an intermediate language representation of the program may be created, such as assembly **630**. The assembly **630** may comprise the intermediate language component **650** and metadata **642**. Application designs may be able to be validated before deployment.

[0066] The various techniques described herein may be implemented in connection with hardware or software or, where appropriate, with a combination of both. Thus, the methods and apparatus described herein, or certain aspects or portions thereof, may take the form of program code (i.e., instructions) embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other machine-readable storage medium, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing aspects of the subject matter disclosed herein. As used herein, the term "machine-readable storage medium" shall be taken to exclude any mechanism that provides (i.e., stores and/or transmits) any form of propagated signals. In the case of program code execution on programmable computers, the computing device will generally include a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device. One or more programs that may utilize the creation and/or implementation of domain-specific programming models aspects, e.g., through the use of a data processing API or the like, may be implemented in a high level procedural or object oriented programming language to communicate with a computer system. However, the program(s) can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language, and combined with hardware implementations.

[0067] Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims.

What is claimed:

1. A system comprising:

at least one processor:

a memory connected to the at least one processor: and

a compiler component that when loaded into the at least one processor causes the at least one processor to:

perform static type checking in a static type system on a plurality of types and on a plurality of values defined in a plurality of modules in a runtime system that dynamically composes software programs from at least one module of the plurality of modules.

2. The system of claim **1**, further comprising:

wherein at least one of the plurality of types is not a global type.

3. The system of claim **1**, further comprising:

wherein at least one of the plurality of values is not a global value.

4. The system of claim **1**, further comprising:

a compiler component that when loaded into the at least one processor causes the at least one processor to:

use a language mechanism to establish a local name binding to a first module of the plurality of modules within content of a second module of the plurality of modules.

5. The system of claim **1**, further comprising:

a compiler component that when loaded into the at least one processor causes the at least one processor to:

map type nomenclature in the static type system to value nomenclature in the runtime system.

6. The system of claim **1**, further comprising:

a compiler component that when loaded into the at least one processor causes the at least one processor to:

perform type checking at compile time by establishing an environment that corresponds to a runtime instance of a plurality of possible runtimes of the program.

7. The system of claim **1**, further comprising:

a compiler component that when loaded into the at least one processor causes the at least one processor to:

determine runtime dependencies of a first module of the plurality of modules to at least a second module of the plurality of modules at compile time.

8. A method comprising:

receiving by a processor of a software development computer, program source code representing a plurality of modules that at runtime comprise a disjoint module universe;

analyzing content of the program source code representing the plurality of modules;

creating an environment wherein types and values from different disjoint universes of a dynamically composing runtime system are used to check the types of expressions described by imports and exports associated with the different disjoint universes.

9. The method of claim **8**, further comprising:

mapping type nomenclature in a static type system to value nomenclature in the dynamically composing runtime system.

10. The method of claim **8**, further comprising:

using a language mechanism comprising an import statement to establish a local name binding to a first module of the plurality of modules within content of a second module of the plurality of modules.

11. The method of claim **8**, further comprising:

using an explicit declarative import within a first module of the plurality of modules to give a locally scoped name to an entity defined in a second module of the plurality of modules.

12. The method of claim **8**, further comprising:

arranging a static type system to align with values of the dynamically composing runtime system, such that a name used to refer to an object at runtime is identical to a name used to refer to a type of the object in the static type system.

13. The method of claim **8**, further comprising;

performing type checking at compile time by establishing an environment that corresponds to a runtime instance of a plurality of runtime instances of a program.

14. A computer-readable storage medium comprising computer-readable instructions which when executed cause at least one processor of a computing device to:

receive by a processor of a software development computer, program source code representing a plurality of modules that at runtime comprises a disjoint module universe;

analyze content of the program source code representing the plurality of modules;

create an environment wherein types and values from different disjoint universes are used to check type expressions using described imports and exports associated with the different disjoint module universes.

15. The computer-readable storage medium of claim **14**, comprising further computer-readable instructions which when executed cause the at least one processor to:

arrange a static type system to align with runtime values of the disjoint module universe, such that a name used to refer to an object at runtime is identical to a name used to refer to a type of the object in the static type system.

16. The computer-readable storage medium of claim **14**, comprising further computer-readable instructions which when executed cause the at least one processor to:

perform type checking at compile time by establishing an environment that corresponds to a runtime instance of a plurality of possible runtime instances of a program.

17. The computer-readable storage medium of claim **14**, comprising further computer-readable instructions which when executed cause the at least one processor to:

use a language mechanism to establish a local name binding to a first module of the plurality of modules within content of a second module of the plurality of modules.

18. The computer-readable storage medium of claim **14**, comprising further computer-readable instructions which when executed cause the at least one processor to:

redetermine dependencies between modules of the plurality of modules at compile time.

19. The computer-readable storage medium of claim **14**, comprising further computer-readable instructions which when executed cause the at least one processor to:

use a language mechanism comprising an import statement to establish a local name binding to a first module of the plurality of modules within content of a second module of the plurality of modules.

20. The computer-readable storage medium of claim **14**, comprising further computer-readable instructions which when executed cause the at least one processor to:

receive program source code written in TypeScript.

* * * * *