(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
27 February 2020 (27.02.2020)

WIPO | PCT

(10) International Publication Number
WO 2020/041411 A2

(51) International Patent Classification:
G06Q 10/08 (2012.01)

(21) International Application Number:
PCT/US2019/047411

(22) International Filing Date:
21 August 2019 (21.08.2019)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/720,665 21 August 2018 (21.08.2018) US
62/838,157 24 April 2019 (24.04.2019) US

(71) Applicant: **BCDB, INC.** [US/US]; 2150 Shattuck Avenue, Penthouse, Berkeley, California 94607 (US).

(72) Inventor: **LEY, Clemens**; 247 4th Street Unit 101, Oakland, California 94607 (US).

(74) Agent: **SERBIN, Gary**; Koffsky Schwalb LLC, 349 Fifth Avenue, Suite 733, New York, New York 10016 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report (Rule 48.2(g))*

(54) Title: OBJECT ORIENTED SMART CONTRACTS FOR UTXO-BASED BLOCKCHAINS

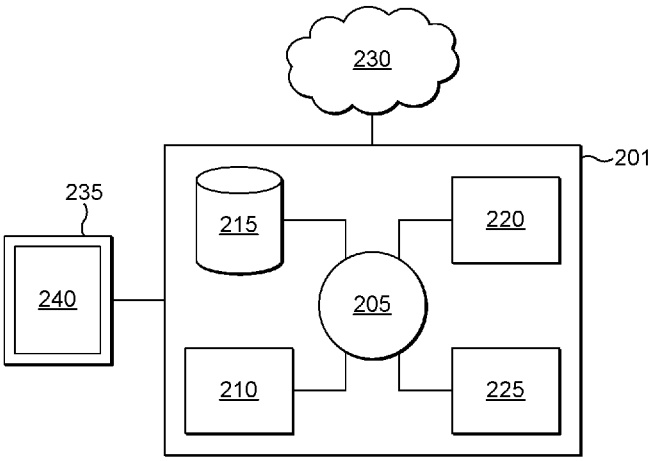

FIG. 2

(57) Abstract: Disclosed is method and system for turning existing object-oriented programming languages into smart contract languages without introducing new syntactic features. The invented method and system provide a protocol that enables storing a history of computations on a decentralized computer network, such as a UTXO-based blockchain system, for any object-oriented computer language. The invented method and system further provide for storing and updating data on blockchains, where such blockchains may be used in cryptocurrency applications and for smart contracts.

WO 2020/041411 A2

OBJECT ORIENTED SMART CONTRACTS FOR UTXO-BASED BLOCKCHAINS

REFERENCE TO PRIOR APPLICATIONS

[0001] This application claims the benefit of the following two applications, each of which is hereby incorporated by reference in its entirety:

1) U.S. Provisional Application Serial No. 62/720,665 filed on August 21, 2018; and

2) U.S. Provisional Application Serial No. 62/838,157 filed on April 24, 2019.

No disclaimer is made to any portions of the above two provisional applications.

FIELD OF THE DISCLOSURE

[0002] The present invention relates to an improved method storing and updating data structures in blockchains, where such blockchains may be used in cryptocurrency applications.

BACKGROUND

[0003] Cryptocurrency, also generally referred to as tokens, coins, digital tokens, digital coins, crypto-tokens, and crypto-coins, may be digital or virtual units of credit (e.g., monetary credit) that are created, stored, transferred, and/or otherwise managed in a decentralized manner, that is, without central management (e.g., by a governmental entity or other regulatory authority). For example, cryptocurrency may be implemented on blockchains or other distributed database structures located on decentralized computer network/system. Digital coins may each be implemented on an independent blockchain database or platform. Digital tokens may be implemented or hosted on an existing blockchain database or platform, such as via implementing a smart contract on the existing blockchain database or platform. Digital tokens may have various functions, such as for use as a currency, an asset (or representation thereof), a security, and an evidence or proof (e.g., of access or membership, etc.). A token may be issued under a token contract implementing a token standard (specification), which can set and define the parameters and rules of token manipulation, user interaction, and other functions and transactions. To date, a vast

majority of digital tokens have been implemented by executing smart contracts on the Ethereum platform under Ethereum Request for Comments (ERC) standards, such as the ERC-20, ERC-223, ERC-621, ERC-721, and the like.

[0004] There are two components to a cryptocurrency: a data structure called a blockchain and a consensus algorithm that is used to update the blockchain.

[0005] Blockchain data structures are processed by a decentralized computer network.

[0006] Figure 1 illustrates one embodiment of the computer system 100 of the present invention. A user at a local computer 103, or the local computer 103 on its own, may interact with a decentralized computer network 101 through a communication network 110, such as the Internet. The decentralized computer network 101 includes a distributed database.

[0007] Figure 2 illustrates an embodiment of a computer 201 of the decentralized computer network 101 in Figure 1. The computer 201 can be configured to implement any computing system disclosed in the present application. For example, the computer 201 may be programmed to or otherwise configured to implement blockchain structures, and digital tokens and/or smart contracts thereon. The computer 201 can be an electronic device of a user or a computer system that is remotely located with respect to the electronic device. The electronic device can be a mobile electronic device. The computer can be a server connected to a plurality of electronic devices of a plurality of users (e.g., participants of the information exchange platform).

[0008] The computer 201 includes a central processing unit (CPU, also "processor" and "computer processor" herein) 205, which can be a single core or multi core processor, or a plurality of processors for parallel processing. The CPU can be the processor as described above. The computer 201 also includes memory or memory location 210 (e.g., random-access memory, read-only memory, flash memory), electronic storage unit 215 (e.g., hard disk), communication interface 220 (e.g., network adapter) for communicating with one or more other systems, and peripheral devices 225, such as cache, other memory, data storage and/or electronic display adapters. In some cases, the communication interface may allow the computer to be

in communication with another device such as the imaging device or audio device. The computer may be able to receive input data from the coupled devices for analysis. The computer 201 can include or be in communication with an electronic display 235 that comprises a user interface 240, such as a scanning interface. The memory 210, storage unit 215, interface 220, the display 235, and peripheral devices 225 are in communication with the CPU 205 through a communication bus (solid lines), such as a motherboard. The storage unit 215 can be a data storage unit (or data repository) for storing data. The computer 201 can be operatively coupled to a communication network 230 with the aid of the communication interface 220. The communication network 230 can be the Internet, an internet and/or extranet, or an intranet and/or extranet that is in communication with the Internet. The communication network 230 in some cases is a telecommunication and/or data network. The communication network 230 can include one or more computer servers, which can enable distributed computing, such as cloud computing. The communication network 230, in some cases with the aid of the computer 201, can implement a peer-to-peer communication network, which may enable devices coupled to the computer 201 to behave as a client or a server.

[0009] Each of the components of the invented computer system 100 (local computer 103 and computers of the decentralized computer network 101) in Figure 1 may be operatively connected to one another via one or more networks 110 or any communication links that allows transmission of data from one component to another. For example, the respective hardware components, if any, may comprise network adaptors allowing unidirectional and/or bidirectional communication with one or more networks. For instance, the user devices and the distributed database (e.g., nodes or components thereof) may be in communication via the one or more networks 110 to transmit and/or receive relevant data.

[00010] The distributed database of the decentralized computer network 101 may be a blockchain. The blockchain may be a distributed ledger enabling the storage of data records as unique blocks connected by one or more secure links. The decentralized computer network containing a blockchain may be cryptographically

secured. A given block in a blockchain may associate transaction data with a timestamp. In the blockchain, duplicate data can be recorded as unique blocks instead of as identical copies of data. A given block may comprise data of a previous block to the given block (e.g., wherein the data of the previous block is hashed), making the blockchain essentially immutable, as data once recorded in a block in the distributed ledger cannot be modified or removed without triggering inconsistency with the linked blocks. This immutable property can provide particular benefits to implementing digital tokens, such as to prevent forgery or other frauds in processing digital credits. A blockchain may comprise or implement one or more smart contracts (implementing token standards), as described elsewhere herein.

[00011] For example, the distributed database of the decentralized computer network 101 may store one or more definitions of what constitutes a valid token on the blockchain (or otherwise definitions for token issuance). The definitions may be constructed in a variety of formats, such as, but not limited to a programming language (e.g., Javascript, C, assembly code, etc.), any other formal language such as mathematics (*e.g.*, set theory, lambda calculus, etc.), natural language, other languages, or a combination thereof. The definitions may be stored on the blockchain via different methods. In some instances, the definitions may be stored on the distributed database directly. Alternatively, or in addition, the definitions may be indirectly stored, such as via anchoring, to the distributed database. For example, the definitions can be stored with an external server, such as in one or more external databases, and an anchor (e.g., hash, universal resource locator (URL), other reference, etc.) to the definitions can be stored on the distributed database to store the definitions by reference. The definitions can be stored by reference via any other method. The definitions or anchors thereof may be stored as plain text in the distributed database. The definitions or anchors thereof may be stored on the distributed database in unencrypted form. Alternatively, the definitions or anchors thereof may be stored on the distributed database in encoded form, encrypted form, compressed form, or a combination thereof. In some instances, the definitions or anchors thereof may be stored as standalone text. In some instances, the definitions or

anchors thereof may be stored as part of, and/or in the form of, output scripts, OP_RETURN scripts (script opcode to mark a transaction output as invalid), multisignature (multisig) addresses, P2SH scripts, and other formats.

[00012] The distributed database may be stored in one or more nodes. The one or more nodes may be distributed in one or more computer systems or devices, such as those described herein. When the distributed database is updated by one of the nodes in the one or more nodes, it may be updated in each node of the one or more nodes, such as via the network 110. The distributed database may be publicly accessed by any node. Beneficially, the distributed nature of the database may be managed without having a central authority present.

[00013] The distributed database of the decentralized computer network 101 may implement one or more smart contracts, including the token contracts. A contract may implement a token standard. In some instances, the smart contracts may be created, modified, viewed, or otherwise accessed via a user interface. The user interface may be a graphical user interface (GUI). A user interface, such as the interface, may be provided by an external server. The external server may or may not be specific to the distributed database. For example, an external server specific to the distributed database may provide services customized for interacting with the specific distributed database, such as by offering services for creating, modifying, viewing, or otherwise accessing smart contracts implemented, or for implementation, on the distributed database. Such services may be used via an interface offered by the external server. For example, the interface may be a web-based interface, mobile interface, application interface, and/or executable program interface. The external server may store relevant information and data in one or more external databases. In some instances, such relevant information and data can include user information data and distributed database information data. One or more template standards may be stored and accessed from the one or more external databases coupled to the external server.

[00014] A server (e.g., external server) may include a web server, an enterprise server, or any other type of computer server, and can be computer programmed to

accept requests (e.g., HTTP, or other protocols that can initiate data transmission) from a computing device (e.g., user device, other servers) and to serve the computing device with requested data. In addition, a server can be a broadcasting facility, such as free-to-air, cable, satellite, and other broadcasting facility, for distributing data. A server may also be a server in a data network (e.g., a cloud computing network).

[00015] A server may include various computing components, such as one or more processors, one or more memory devices storing software instructions executed by the processor(s), and data. A server can have one or more processors and at least one memory for storing program instructions. The processor(s) can be a single or multiple microprocessors, field programmable gate arrays (FPGAs), or digital signal processors (DSPs) capable of executing particular sets of instructions. Computer-readable instructions can be stored on a tangible non-transitory computer-readable medium, such as a flexible disk, a hard disk, a CD-ROM (compact disk-read only memory), and MO (magneto-optical), a DVD-ROM (digital versatile disk-read only memory), a DVD RAM (digital versatile disk-random access memory), or a semiconductor memory. Alternatively, the methods can be implemented in hardware components or combinations of hardware and software such as, for example, ASICs, special purpose computers, or general purpose computers. In some instances, the external server 105 may support code editing.

[00016] The one or more external databases may utilize any suitable database techniques. For instance, structured query language (SQL) or "NoSQL" database may be utilized for storing data. Some of the databases may be implemented using various standard data-structures, such as an array, hash, (linked) list, struct, structured text file (e.g., XML), table, JavaScript Object Notation (JSON), NOSQL and/or the like. Such data-structures may be stored in memory and/or in (structured) files. In another alternative, an object-oriented database may be used. Object databases can include a number of object collections that are grouped and/or linked together by common attributes; they may be related to other object collections by some common attributes. Object-oriented databases perform similarly to relational databases with the exception that objects are not just pieces of data but may have other types of

functionality encapsulated within a given object. Also, the database may be implemented as a mix of data structures, objects, and relational structures. Databases may be consolidated and/or distributed in variations through standard data processing techniques. Portions of databases (e.g., tables) may be exported and/or imported and thus decentralized and/or integrated.

[00017] The decentralized computer network 101 may communicate with one or more local devices 103 (e.g., local computer, user computer, etc.). In some cases, the decentralized computer network 101 may communicate with a large number of computer terminals of a large plurality of different users or entities. A user device may correspond to a node in the database of decentralized computer network 101. A user device may be a computing device configured to perform one or more operations consistent with the disclosed embodiments. For example, the user devices 103 may interact with the distributed database of the decentralized computer network 101 by requesting and obtaining data via the network 110. A user device 103 may be used to broadcast a blockchain transaction, such as a token issuance transaction, smart contract issuance transaction, user creation transaction, and/or token transfer transaction. A user device may be used to store data in the distributed database of decentralized computer network101. In another example, a user device may be used to access a user interface. For example, the user interface may create, modify, or otherwise access a token standard and/or smart contract via the user interface. A graphical user interface may include graphical elements such as buttons or other user indicators that may be selected by the user using a mouse, keyboard, other user control device, or a bodily part or other tool (e.g., for GUIs supported on a touch-sensitive display) for inputting data or otherwise viewing outputs and other intermediary visualizations.

[00018] Examples of user devices may include, but are not limited to, mobile devices, smartphones/cellphones, tablets, personal digital assistants (PDAs), laptop or notebook computers, desktop computers, media content players, television sets, video gaming station/system, virtual reality systems, augmented reality systems, microphones, or any electronic device configured to enable the user to visualize data

or other outputs, for example. The user device may be a handheld object. The user device may be portable. The user device may be carried by a human user. In some cases, the user device may be located remotely from a human user, and the user can control the user device using wireless and/or wired communications.

[00019] The user device may include a communication unit, which may permit the communications with one or more other components in the decentralize computer network 101. In some instances, the communication unit may include a single communication module, or multiple communication modules. In some instances, the user device may be capable of interacting with one or more components in the network environment using a single communication link or multiple different types of communication links.

[00020] A user device may include one or more processors that are capable of executing non-transitory computer readable media that may provide instructions for one or more operations consistent with the disclosed embodiments. The user device may include one or more memory storage devices comprising non-transitory computer readable media including code, logic, or instructions for performing the one or more operations.

[00021] In some embodiments, users may utilize the one or more user devices to interact with the external server by way of one or more software applications (i.e., client software) running on and/or accessed by the user devices, wherein a user device 103 forms a client relationship with the external server. For example, the user device, such as a local computer, 103 may run dedicated mobile applications associated with the external server and/or utilize one or more browser applications to access external server's interface. In turn, the external server may deliver information and content to the user device 103 related to the distributed database of the decentralized computer network 101, for example, by way of one or more web pages or pages/views of a mobile application. As described elsewhere herein, the user device 103 may include several user devices that communicate amongst each other to form nodal relationships (e.g., as opposed to specific client-server relationships) within the computer system 100.

[00022] In some embodiments, the client software (i.e., software applications installed on the local device 103) may be available either as downloadable mobile applications for various types of mobile devices. Alternatively, the client software can be implemented in a combination of one or more programming languages and markup languages for execution by various web browsers. For example, the client software can be executed in web browsers that support JavaScript and HTML rendering, such as Chrome, Mozilla Firefox, Internet Explorer, Safari, and any other compatible web browsers. The various embodiments of client software applications may be compiled for various devices, across multiple platforms, and may be optimized for their respective native platforms.

[00023] The local device may include a display. The display may be a screen. The display may or may not be a touchscreen. The display may be a light-emitting diode (LED) screen, OLED screen, liquid crystal display (LCD) screen, plasma screen, or any other type of screen. The display may be configured to show a user interface (UI), such as a graphical user interface (GUI) rendered through an application (e.g., via an application programming interface (API) executed on the user device). The GUI may show graphical elements that permit a user to view a visualization of the token standard, the smart contract, and the like. The GUI may also allow a user to provide inputs and view outputs. In some instances, the UI may be web-based. For example, the local device may also be configured to display webpages and/or websites on the Internet.

[00024] In some cases, the system environment may comprise a cloud infrastructure. One or more virtual systems such as Docker systems may be utilized in the network for allowing the multiple users or user devices to interact. In such cases, each user device can be considered to be a processing environment that is being used by a manager/participant as part of a financial activity, for example. The plurality of user devices may comprise heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs). For example, the user device may allow one or more users to access applications through either a thin client interface, such as a web browser or program interface. The plurality of user devices may comprise any

general purpose or special purpose computing system environments or configurations. Examples of well-known computing systems, environments, and/or configurations that may be suitable for use with computer system/server include, but are not limited to, personal computer systems, server computer systems, thin clients, thick clients, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer electronics, network PCs, minicomputer systems, mainframe computer systems, distributed cloud computing environments that include any of the above systems or devices, and the like.

[00025] User devices may be associated with one or more users. In some embodiments, a user may be associated with a unique user device. Alternatively, a user may be associated with a plurality of user devices. A user as described herein may refer to an individual or a group of individuals or any user who is interacting with the distributed database of the decentralized computer network 101.

[00026] The one or more communication networks 110 may be a set of communication pathways between the user devices (e.g., local computers) 103, and other components of the communication network. A communication network may comprise any combination of local area and/or wide area networks using both wireless and/or wired communication systems. For example, the communication network may include the Internet, as well as mobile telephone networks. In one embodiment, the network uses standard communications technologies and/or protocols. Hence, the network may include links using technologies such as Ethernet, 802.11, worldwide interoperability for microwave access (WiMAX), 2G/3G/4G or Long Term Evolution (LTE) mobile communications protocols, Infra-Red (IR) communication technologies, and/or Wi-Fi, and may be wireless, wired, , asynchronous transfer mode (ATM), InfiniBand, PCI Express Advanced Switching, or a combination thereof. Other networking protocols used on the network can include multiprotocol label switching (MPLS), the transmission control protocol/Internet protocol (TCP/IP), the User Datagram Protocol (UDP), the hypertext transport protocol (HTTP), the simple mail transfer protocol (SMTP), the file transfer protocol (FTP), and the like. The data exchanged over the network can be represented using technologies and/or formats

including image data in binary form (e.g., Portable Networks Graphics (PNG)), the hypertext markup language (HTML), the extensible markup language (XML), etc. In addition, all or some of links can be encrypted using conventional encryption technologies such as secure sockets layers (SSL), transport layer security (TLS), Internet Protocol security (IPsec), etc. In another embodiment, the entities on the communication network can use custom and/or dedicated data communications technologies instead of, or in addition to, the ones described above. The network may be wireless, wired, or a combination thereof.

[00027] Users running the decentralized computer network 101 are called miners or validators. This group of users runs consensus protocols, such as proof or work, proof of stake, delegated proof of stake, (practical) byzantine fault tolerance and others, to maintain consensus over the state of the blockchain. The decentralized computer network keeps track of the creation of cryptographic coins as well as their transfer from user to user.

[00028] Some blockchains store coins in data structures called unspent transaction outputs (UTXO). In such UTXO-type blockchains, each UTXO contains a function called a spending condition. In a UXTO-type blockchain platform, the decentralized computer network would implement a protocol guaranteeing that a system user can only send the coins to another UTXO if the sender can provide an input that satisfies the spending condition. To send coins, a sender would create a data structure called a transaction (a request) that includes two lists: the first list of inputs, each specifying which UTXO to send from and a parameter to the spending condition. The second list includes information enabling creation of new UTXO that contain the coin(s) being sent.

[00029] Spending conditions are encoded using a scripting language, which may either be Turing-complete or not Turing-complete ("Turing-incomplete.")

[00030] If a system uses a Turing-complete scripting language, it could become susceptible to an attack in which a malicious user broadcasts a transaction with a non-terminating spending condition. Every miner has to execute all spending conditions in the blockchain to ensure that a transaction is only included in the blockchain if the

parameters in the inputs satisfy the respective spending conditions. If the scripting language used for defining spending conditions is Turing-complete, however, a miner checking such malicious output may be caught in a non-terminating computation. As a result, the entire network may become unresponsive, and a hard fork would be required to remove the malicious transaction from the blockchain.

[00031] One prior art approach for avoiding such an attack, while using a Turing-complete scripting language, is to implement an Ethereum-type blockchain platform, which charges a user for every computational step in the spending condition. This makes non-terminating computation infinitely expensive to the attacker and, therefore, impractical. However, a lot of record keeping is required to make this approach to building blockchains work: a second token, called "gas," is needed with which users pay the miners for performing the computation; two kinds of accounts (externally owned accounts and contract accounts) are needed as opposed to one kind used in a UTXO-based blockchain discussed below (private-public key pairs)); four different types of data structures (state trie, storage trie, transaction trie, recipes trie) are needed as opposed to one in UTXO-based model (the blockchain). This complexity makes it hard to reason about the system, which in turn makes it difficult and costly to develop secure applications. For example, even experienced programmers may have difficulty writing secure smart contracts in a Ethereum-type blockchain, which can result in security breaches and hacks.

[00032] Another drawback of the Ethereum-type blockchain platform is the cost of running smart contracts on it. This is because all miners need to perform every computation redundantly. Here, miners communicate through a byzantine fault tolerant protocol which adds extra overhead. Overall, running a computation on Ethereum is roughly 1,000,000 times more expensive than the same computation on consumer hardware.

[00033] At the same time, existing prior art systems using Turing-incomplete scripting languages, while not being susceptible to the type of an attack described above, have their own drawbacks. For example, an approach called UTXO-based

blockchain uses a scripting language that cannot express all possible spending conditions, limiting the use of UTXO-based blockchains in real world applications.

[00034] Some advanced applications on top of UTXO-type blockchains use an approach called colored-coin approach. Colored-coin-based protocols typically support a class of smart contracts called "tokens" that can be used to generate digital entities that are scarce: new tokens can only be created under predefined conditions, making their duplication difficult and very costly.

[00035] The idea is to record meta data about token creation and token transfer transaction. Each colored-coin protocol is accompanied by a documentary standard (specification) that defines a fixed set of valid token creation and token transfer transactions. Users can broadcast these standardized transactions to create a provenance trail of token transfer events. The documentary specifications define ways in which transactions can or cannot be combined. For example, to achieve scarcity, a specification may prohibit a token transfer transaction that creates more tokens than it consumes.

[00036] Miners are unaware of the colored-coin standard being used and include a token transaction without checking its validity with respect to the standard. Although this has the advantage of miners not being required to do any additional computational work, the disadvantage is that users cannot assume that tokens recorded on a blockchain are valid. What allows this type of system work, however, is that the blockchain contains enough information to enable a user to determine the token's validity. To do so, for every token the user is offered, the user checks the history of token transfers. If the history contains only valid transfers, then the token is deemed valid and can safely be accepted.

[00037] Although the colored-coin approach provides enhanced security and reliability, it has at least two drawbacks. The first drawback is a lack of expressiveness. For example, while most colored-coin protocols support data structures that are mappings from user IDs to basic data types (such as numbers, strings, Booleans), they do not support operations on complex (non-basic) data structures.

[00038] The second drawback of colored-coins protocols is their limiting of permitted data updates. Each colored-coin standard supports a fixed set of about a handful of token creation and transfer types. This precludes applications requiring a different kind of data updates. As a result, although the colored-coin approach has its advantages, it works only for a small number of applications.

[00039] Accordingly, there is a need for a protocol for smart contracts that overcomes the disadvantages of the prior art, such as the Ethereum protocol and the colored-coin protocol.

[00040] For example, there is a need for a blockchain protocol that supports every computational token transfer type on UTXO-based blockchains.

[00041] There is also a need for an alternative semantic protocol that enables storing a history of computations on a UTXO-based blockchain for any object-oriented computer language, such as the Classroom Object-Oriented Language ("COOL"). This history can be used for audit purposes and/or for proving properties of a system to an outside observer.

[00042] There is also a need for cryptocurrency approach that works on non basic, nested data structures independent of the consensus algorithm used.

[00043] There is also a need for a method and system for storing and updating on a UTXO-based blockchain platform non-basic data structures. Particularly, there is a need for a method and system for storing and updating on such UTXO-based blockchain platform non-basic, nested data structures. More particularly, there is a need for a method and system for storing and updating non-basic, nested data structures on UTXO-based blockchain platforms.

[00044] Recognized herein is a need for systems and methods for digital token (which is a particular kind of smart contracts) implementations on various distributed database structures that address at least the above mentioned problems and objectives.

SUMMARY

[00045] The present invention overcomes the problems of the prior art. For example, whereas each colored coin standard supports a fixed set of supported token

transfer types, the present invention supports every computable transfer type. Instead of standards documents, the invented system uses object-oriented programs as specifications. For a given object-oriented language, we define an alternative semantics that stores a trace of a computation on a blockchain. Traces can be valid or invalid, but like in the colored coins approach, miners are unaware of the meaning of the trace and will include transactions encoding invalid traces. However, also as in the colored coin approach, a user will be able to determine from information on the blockchain whether the trace is valid. If the trace is valid it can be used to compute a value.

[00046] In the same way that colored coins store tokens in UTXOs, the approach of present invention stores the current state of a computation in multiple utxos. Values can be updated by executing function calls. Each function call corresponds to a transaction that intuitively speaking spends the output storing the old value into an output storing a new value. The spending condition of the utxo restricts who can call a function that updates the value in the utxo. This naturally induces a notion of data ownership where only the owner of some data has permission to update the value. A new owner can be assigned by calling a function. Additionally, coins can be stored together with a value in an utxo and these coins can be transferred from user to user via function calls in the same way that the date is updated.

[00047] In Ethereum, miners provide consensus over the value of a computation. Miners in Bitcoin cannot do that, but they do provide consensus over the state (for example who owns how many coins). In the present invention, a trace pf the configurations of the computation are stored. Users can analyze a trace and gain consensus over the value of the computation. This makes the invented system trustless.

[00048] The present invention has the following advantages:

- Smart contracts can be built on top of utxo based blockchains. Despite the fact that these represent a majority of cryptocurrencies, currently, no other general-purpose, Turing complete, smart contract solution exists for either of them.

- Existing object-oriented programming languages can be turned into smart contract languages without introducing new syntactic features. The inventions thereby adds smart contract capabilities to existing ecosystems of developers, tools, and theoretical underpinnings that have been developed by large communities over many years.
- Like in the colored coins approach, computational work is performed by the users and not by the miners. This leads to immense cost-saving, because of the high cost of performing computation by miners, where as the electricity cost for performing a computation on a local computing device can be considered negligible for most applications (users "pay" with their time).
- All previous states of a system can be recovered from the blockchain. All updates are time-stamped and cryptographically signed. This information can be used to audit a system in hindsight.
- The system allows multiple users to manipulate the same data without requiring trust between the users. This makes it easy to build applications that span multiple organizations. It also makes it possible for service providers to guarantee properties of their service to their users.

[00049] The present invention provides method of generating an instruction in a first object-oriented computer language for operating in a decentralized computer network from an instruction in a second object-oriented computer language that can operate on a local (centralized) computer, where a semantic of the second object-oriented computer language can be expressed by a rule so, S, E, ⊢ e: v, S' □□with ⊢ denoting a semantic relation said rule meaning that in a context where a variable self refers to an object *so*, where a data store on the centralized computer is S, and where an execution environment is E, an expression *e* evaluates to an object *v* at a second data store *S'* in the centralized computer, the method comprising: replacing a centralized (local) storage update command S[v/l], which updates an at least one location l by an object v, by a command to (a) broadcast to the decentralized computer

network an at least one transaction that spends at least one unspent-transaction-output corresponding to said location l into a new output that stores at least one of (i) said object v and (ii) information enabling said object v to be determined; and (b) spending at least one additional unspent-transaction-output corresponding to a second location l' in said execution environment E or said object so, wherein said generated instruction enables operation of a smart contract.

[00050] The present invention further provides that the instruction in said first object-oriented computer language is a storage-update instruction, and wherein said storage-update instruction either immediately precedes a recursive call to a first semantic relation or is located at an end of a body of said semantic rule of said first object-oriented computer language.

[00051] The present invention further provides that the instruction in said first object-oriented computer language is an instruction for storing a data structure *D* at an unspent memory location *OUT* on the decentralized computer network, said data structure *D* further including a data structure *D'*, where said data structure D' further includes a data structure *D''*, said method comprising the steps of:

(a) receiving at the local computer a request to store the data structure *D* at an unspent memory location *OUT* on the decentralized computer network;

(b) creating at the local computer a request *TX*, to the decentralized computer network, to spend the unspent memory location *OUT* and to generate a new unspent memory location *OUT'* on the decentralized computer network;

(c) using the local computer to sign and broadcast said request *TX* to the decentralized computer network;

(d) receiving said request *TX* at the decentralized computer network;

(e) using the decentralized computer network to determine validity of said request *TX* according to a protocol of the decentralized computer network; and

(f) if said request *TX* is determined to be valid, performing the steps of

(i) spending the unspent memory location *OUT* on the decentralized computer network and generating said new unspent memory locations *OUT'* on the decentralized computer network; and

(ii) repeating steps (a) through (f)(i) above while using said data structure *D'* in place of said data structure *D* and while using said memory location *OUT'* in place of said memory location *OUT*.

[00052] The present invention further operation on a decentralized computer network that is an unspent-transaction-outputs blockchain.

[00053] The present invention further enables a source code of a program written in one of said first computer language and said second computer language to be stored in a transaction, for example an object creation stransaction.

[00054] The present invention provides a computer system comprising a local computer, the local computer comprising:

a) a memory for storing computer instruction and data; and

b) a processor operatively coupled to said memory, said processor capable of generating an instruction in a first object-oriented computer language for operating in a decentralized computer network from an instruction in a second object-oriented computer language that can operate on the local computer, where a semantic of the second object-oriented computer language can be expressed by a rule

so, S, E, ⊢ e: v, S'

with ⊢ denoting a semantic relation said rule meaning that in a context where a variable self refers to an object *so*, where a data store on the local computer is S, and where an execution environment is E, an expression *e* evaluates to an object *v* at a second data store *S'* in the local computer, said method comprising:

replacing a centralized storage update command S[v/l], which updates an at least one location l by an object v, by a command to

(a) broadcast to the decentralized computer network an at least one transaction that spends at least one unspent-transaction-output corresponding to said

location l into a new output that stores at least one of (i) said object v and (ii) information enabling said object v to be determined; and

(b) spending at least one additional unspent-transaction-output corresponding to a second location l' in said execution environment E or said object so;

wherein said generated instruction enables operation of a smart contract.

BRIEF DESCRIPTION OF THE FIGURES

[00055] The accompanying figures, where like reference numerals refer to identical or functionally similar elements throughout the separate views, together with the detailed description below, are incorporated in and form part of the specification, and serve to further illustrate embodiments of concepts that include the claimed invention and explain various principles and advantages of those embodiments.

[00056] Figure 1 is a computer system in accordance with some embodiments of the present invention.

[00057] Figure 2 shows a computer on a decentralized computer network in accordance with some embodiments of the present invention.

[00058] Figure 3 is a diagram of the transactions that are broadcast when a location in the store of the decentralized computer is updated.

[00059] Figure 4 is a diagram of the transactions that are broadcast when a variable assignment Id<-e is evaluated in a blockchain semantics in accordance with some embodiments of the present invention.

[00060] Figure 5 is a diagram of the transactions that are broadcast when an expression of the form let Id: T <- e in e' is evaluated in a blockchain semantics in accordance with some embodiments of the present invention.

[00061] Figure 6 is a diagram of the transactions that are broadcast when an expression of the form new A is evaluated in semantics in accordance with some embodiments of the present invention.

[00062] Figure 7 is a diagram of the transactions that are broadcast when an expression of the form $e_0.f(e_0, ..., e_0)$ is evaluated in a blockchain semantics in accordance with some embodiments of the present invention.

[00063] Figure 8 is a diagram of the transactions that are broadcast when an expression let x <- 1 in y <- 2 in x<- y is evaluated in a blockchain semantics in accordance with some embodiments of the present invention.

[00064] Figure 9 is a diagram of the transactions that are broadcast when an expression new A is evaluated in a blockchain semantics in accordance with some embodiments of the present invention.

[00065] Figure 10 is a diagram of the transactions that are broadcast when an expression new B is evaluated in a blockchain semantics in accordance with some embodiments of the present invention.

[00066] Figure 11 is a diagram of the transactions that are broadcast when an expression let c <- new Counter in a.inc() is evaluated in a blockchain semantics in accordance with some embodiments of the present invention.

[00067] Figure 12 is a diagram of a transaction context in accordance with some embodiments of the present invention.

[00068] Figure 13 is a diagram of a process of plugging a context into a hole in accordance with some embodiments of the present invention.

[00069] Figure 14 is a diagram of a process of plugging an empty context into a hole in accordance with some embodiments of the present invention.

[00070] Figure 15 is a diagram of context [Ctx-Assign] for semantic rule [B-Assign] of a blockchain semantics in accordance with some embodiments of the present invention.

[00071] Figure 16 is a diagram of context [Ctx-New] for semantic rule [B-New] of a blockchain semantics in accordance with some embodiments of the present invention.

[00072] Figure 17 is a diagram of context [Ctx-Let] for semantic rule [B-Let] of a blockchain semantics in accordance with some embodiments of the present invention.

[00073] Figure 18 is a diagram of context [Ctx-Dispatch] for semantic rule [B-Dispatch] of a blockchain semantics in accordance with some embodiments of the present invention.

[00074] Skilled artisans will appreciate that elements in the figures are illustrated for simplicity and clarity. The apparatus and method components have been represented where appropriate by conventional symbols in the drawings, showing only those specific details that are pertinent to understanding the embodiments of the present invention so as not to obscure the disclosure with details that will be readily apparent to those of ordinary skill in the art having the benefit of the description herein.

DETAILED DESCRIPTION

[00075] The following detailed description discloses some embodiments of the present invention. A person skilled in the art, however, will understand that the present invention in not limited to the specific disclosed embodiments, and that other embodiments are within the scope of the present invention.

[00076] Although the present invention can work with any object-oriented programing language, one exemplary embodiment implements the invention using a UXTO-type blockchain and a simple object-oriented language called Classroom Object-Oriented Language ("COOL").

[00077] A UXTO-based cryptocurrency has two elements: a data structure called a blockchain and a consensus algorithm that is used to update the blockchain. The present invention is independent of the consensus algorithm used and can operate on all UTXO-based blockchains regardless of the consensus protocol used.

[00078] In one embodiment of the present invention, a blockchain *B* includes a set of transactions (requests issued to a distributed computer network), where a transaction consists of a list of inputs *ins* and a list of outputs *outs*. Each transaction (request) *tx* and each output *out* in a blockchain *B* has a unique identifier denoted by *Id(tx, B)* and *Id(out, B)*, respectively. Given an identifier id of an object, *B(id)* denotes the object in the blockchain B, therefore $B(Id(tx, B)) = tx$ and $B(Id(out, B)) = out$.

[00079] Below is an exemplary expression for a transaction *tx* having a number of inputs *ins* and outputs *outs*.

```
tx = {
```

ins: [{ old_1, ?$params_1$ }, ..., { old_n, ?$params_n$ }]
outs [{ $condition_1$, $amount_1$, ?$data_1$ }, ... { $condition_m$, $amount_m$, ?$data_m$ }]
}

[00080] An input in *B* stores the id of an output in *B* in a field called oId_i and may also store an array of parameters in a filed called *params*. A transaction *tx* is considered *signed* if each one of its inputs contains an array of parameters; considered *partially signed* if some inputs have parameters; and considered *unsigned* if no input contains parameters. Given a transaction *tx*, *copy*(*tx*) denotes the unsigned transaction that is obtained from *tx* by removing all parameters from the transaction's inputs.

[00081] An *output* includes a function *condition* that maps parameters to a Boolean, a number of coins called *amount*, and optionally some data.

[00082] A partial function that maps a transaction and an *n*-ary spending condition to an array of *n*-1 parameters may be referred to as a wallet *W*. *W* can sign an *input {oId: B(out)}* of a transaction *tx* if *copy(tx)* and the condition c_{out} of the output *out* that is being spent are in the domain of *W*. For example, $sign_W(tx)$ denotes the transaction obtained from *tx* by replacing every unsigned input *{oId: B(out)}* that *W* can sign by *{oId: B(out), params: W(copy(tx), c_{out})}*.

[00083] For example, the expression below describes the process of signing a transaction *tx*, in which a wallet *W* can sign the first but not the second input of *tx*. The first input of *tx* references an output out_0 with spending condition *c*. The unsigned transaction *copy(tx)* is obtained from *tx* by removing a parameter *sig* from the second input. $sign_W(tx)$ has *sig* added back as the parameter of the second input and *W(copy(tx), c)* is added as the parameter of the first input.

out_0 = { amount: ..., condition: c }
tx_0 = { ins: ..., outs: [out_0] }
tx = { ins: [{ old: Id(out_0, B) }, { old: ..., params: [sig] }], outs: ... }
copy(tx) = { ins: [{ old: Id(out_0, B) }, { old: ... }], outs: ... }
$sign_W$(tx) = {
ins: [{ outId: Id(out_0, B), params: W(copy(tx), c) }, { old: ..., params: [sig] }],
outs: ...

}

[00084] We can say that input *in* "spends" output *out* if *in* includes a reference to *out*. We can also say that a transaction *tx* "spends" an output *out* if the *out* is spent by an input of *tx*. We can say that a wallet *W* can "spend" from an output *out* with condition *c* if there is a transaction *tx* such that

$$c(copy(tx), W(copy(tx), c)) = true$$

[00085] A blockchain *B* is valid if:

1. Every transaction *tx* in *B* is fully signed and has a unique id *Id(tx, B)*.
2. If *{ oId: o, params: p }* is an input in *B*, then *B* contains an output *out* such that *B(o) = out*.
3. If *{ oId: o_1, params: p_1 }* and *{ oId: o_2, params: p_2 }* are distinct inputs in *B*, then $o_1 \neq o_2$.
4. If *{ oId: o, params: p }* is an input in *B*, then *copy(tx)* and p satisfy the condition B(o).
5. If a transaction *tx* in *B* has at least one input, then the sum of coins in the outputs of *tx* is not greater than the sum of coins in the outputs spent by *tx*.

[00086] A wallet can broadcast a unsigned transaction *tx* to a blockchain *B* if the expression $B \cup \{ sign_W(TX) \}$ is valid.

[00087] A transaction with no inputs is called a *coinbase transaction*. Such a transaction creates new coins. Note that according to the definition in one preferred embodiment, any *coinbase transaction* is valid in any blockchain.

[00088] In real world blockchains, only distinguished users called miners are permitted to broadcast *coinbase transactions*.

[00089] In addition, the consensus protocol may impose further constraints on how frequently *coinbase transactions* can occur and how many coins they can create. Below is an example of a payment system that can be implemented using a blockchain of the present invention.

[00090] Let *(gen, sign, verify)* be a digital signature scheme. Let *(pubKey, privKey)* be a private-public key pair generated by *gen()*. A transaction TX_0 that has

one output *out* with a condition *c* that maps *(x, y)* to *verify(x, pubKey, y)*. Let *W* be a wallet that maps *(copy(tx), c)* to *[privkey, sign(privKey, copy(tx))]* for every transaction *tx*.

out_0={ condition: (x, y) => verify(x, pubKey, y), amount:a }

tx_0 = { ins: ..., outs: [out_0] }

tx = { ins: [{ outId: $Id_B(out_0)$ }], outs: ... }

copy(tx) = tx

$sign_W$(tx) = { ins: [{ outId: Id(out_0 B), params: [privKey, sign(privKey, copy(tx))] }], outs: ... }

[00091] Because the expression below is true, *tx* is a witness that *W* can spend the output tx_0.

c(copy(tx), W(copy(tx), c)) = c(copy(tx), sign(privKey, copy(tx))) = verify(copy(tx), pubKey, sign(privKey, copy(tx))) = true

[00092] Although the present invention can operate with any object-oriented programming language, in one preferred embodiment, the present invention will be described in the context of the COOL programming language. COOL is a statically typed object-oriented language with a formally defined syntax and semantics that are described in "The Cool Reference Manual," which was authored by Alex Aiken in 1995, and the contents of which are hereby incorporated in full by reference and included in the Appendix section of this Detailed Description. The syntax of Cool is similar to the syntax of existing object-oriented programming languages like Java or Javascript.

[00093] Cool programs are sets of classes written is a syntax that is similar to existing object oriented programming languages like Java or Javascript. For every program P, there are two functions that can be computed from the source code of P: the function class returns the attributes, types and initializations for a given class:

$class(X) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow en)$

The function *implementation* returns the parameters and body of a given method in a class:

$implementation(X, m) = (x_1, \ldots, x_n, e_{body})$

[00094] Most constructs in Cool are expressions and every expression has a value. The basic expressions in Cool are integer constants like 1, 2, 3, strings like 'hello', and the Boolean values *true* and *false*. All values are objects in Cool. Values are defined with respect to a set of *identifiers* Ids and a set of *locations Loc*. Identifiers are atomic syntactic objects that can be used as variables. There are also special identifiers, like *self*. The *values with identifiers Ids and locations in Loc* (denoted $Val_{Ids, Loc}$) is the set containing *void*, *Bool(b)* for $b \in \{true, false\}$, *Int(n)* for integer n, *String(l, s)* for string s of length l, and elements of the form

$C(a_1 = loc_1, \ldots, a_n = loc_n)$

where C is a class name, $a_1, \ldots, a_n \in Ids$, and $loc_1, \ldots, loc_n \in Loc$.

[00095] Each value in Cool has a type. The type of an integer constant *x* is *Int* and its value is *Int(x)*, and the same notation is used for strings and Booleans. The type of an object C(a=l) is C. Each type has a default value: the default D_{Int} of *Int* is *0*, the default D_{String} of *String* is the empty string '', the default D_{Bool} of *Bool* is *false*, and the default of any other type is *void*.

[00096] As mentioned above, the set Ids of identifiers in a program always includes a special variable *self* that refers to the object on which a method was dispatched. We write *Val* instead of $Val_{Ids, Loc}$ when the set of ids and the set of locations are clear from the context.

[00097] The semantics of Cool is defined with respect to a partial function called environment E: Ids -> Loc and a partial function called store S: Loc -> Val. The operational semantics of Cool is defined by rules of the form

so, S, E ⊢ e: v, S'

where so is an object, E is an environment, S and S' are stores, e is an expression, and v is a value. The rule reads: In the context where the special variable self refers to the object so, the store is S, and the environment is E, the expression e evaluates to value v and the new store is S'.

[00098] Changes to the store and environment will be denoted using the following notation: Given a function F:D -> R and elements d, d' ∈ D and r ∈ R, we

define F[r/d] to be a function such that R[r/d](d') = r if d = d' and F[r/d](d') = F(d') otherwise. To allocate memory, we need a way to obtain a new location from a store. Thus, we define *newloc(S)* to be a function that returns a location $l \in Loc$ for a given store S, such that l is not in the domain of S.

[00099] The semantic rules of Cool that are relevant here are discussed below (the other rules can be found in Section 13 of the Cool manual, which is incorporated herein in its entirety). For the examples below, we fix an environment E and a store S_1.

[000100] Variable Assignment

The first step when evaluating Id ← e in so, S_1, E, as defined above is to evaluate expression e to obtain its value v. The side effect of the evaluation is that the store S_1 is updated to a store S_2, however, S_2 might be identical to S_1. Next, the location E(Id) of Id is looked up in the environment. Finally, the store S_2 is updated to store value v at location l. The semantics rule [Assign] of the Cool semantics is shown below.

[000101] [Assign]

so, S_1, E ⊢ e: v, S_2

E(Id) = l

$S_3 = S_2[v/l]$

---

so, S_1, E ⊢ Id ← e: v, S_3

[000102] Variable Definition

When an assignment *let Id: T1* ← e_1 *in* e_2 is evaluated in *so,* S_1, *E*, the expression e_1 is evaluated to value v_1. Then a new location l is obtained S_2, and the value v_1 is stored at location l. Finally, the expression e_2 is evaluated in the environment E' = E[l/id].

[000103] [Let]

so, S_1, E ⊢ e: v, S_2

l = newloc(S_2)

$S_3 = S_2[v/l]$

E' = E[l/Id]

so, S_3, E' ⊢ e': v', S_4

---

so, S_1, E ⊢ let Id: T ← e in e': v', S_4

[000104] Object Construction

When an expression *new A* is evaluated, the class A_0 to be constructed is determined (details of SELF_TYPE can be found in section 4.1 of the Cool manual, which is incorporated herein in its entirety). Next, distinct new locations l_i are obtained from S_1 for every attribute a_i: T_i of *A* and 1 <= i <= n. Location l_i in initialized to the default for type A_i. Finally, the expression $a_1 \leftarrow e_1$; ...; $a_n \leftarrow e_n$; is evaluated in an environment that contains the locations that store the attributes of A_0.

[000105] [New]

A_0 = B if A = SELF_TYPE and so=B(...) and A_0 = A otherwise

class(A_0) = (a_1: A_1 ← e_1, ..., a_n: A_n ← e_n)

l_i = newloc(S_1), for i = 1 ..., n and each l_i is distinct

v_1 = A_0(a_1 = l_1, ..., a_n = l_n)

S_2 = S_1[D_{A_1}/l_1, ..., D_{A_n}/l_n]

v_1, S_2, [a_1: l_1, ..., a_n: l_n] ⊢ {a_1 ← e_1; ...; a_n ← e_n;}: v_2, S_3

---

so, S_1, E ⊢ new A: v_1, S_3

[000106] Method Dispatch

When an expression e_0.f(e_1, ..., e_n) is evaluated in the Cool semantics, the expressions e_1, ..., e_n that are passed as parameters are evaluated to the values v_1, ..., v_n, respectively. Then expression e_0 is evaluated to value v_0. Then one new location l_{x_i} is created and initialized to store v_i for 1 <= i <= n. Then the body e_{n+1} of the method f is evaluated in an environment that contains the locations storing the attributes of the object v_0 on which the function was called, as well as the locations l_{x_1}, ..., l_{x_n} that store the values of the parameters

[000107] [Dispatch]

so, S_1, E ⊢ e_1: v_1, S_2

so, S_2, E ⊢ e_2: v_2, S_3

...

so, S_n, E ⊢ e_n: v_n, S_{n+1}

so, S_{n+1}, E ⊢ e_0: v_0, S_{n+2}

A(a_1 = l_{a_1} , ..., a_m = l_{a_m}) = v_0

(x_1, ..., x_n, e_{n+1}) = implementation(A, f)

l_{x_i} = newloc(S_{n+2}), for i = 1 ... n and each l_{x_i} is distinct

S_{n+3} = S_{n+2}[v_1/l_{x_1} , ..., v_n/l_{x_n}]

v_0, S_{n+3}, [a_1: l_{a_1} , ..., a_m: l_{a_m}, x_1: l_{x_1} , ..., x_n: l_{x_n}] ⊢ e_{n+1}: v_{n+1}, S_{n+4}

---

so, S_1, E ⊢ e_0.f(e_1, ..., e_n): v_{n+1}, S_{n+4}

[000108] As mentioned above, the present invention can convert any object-oriented programming language into a smart contract version of that language. One embodiment of the invention described below converts the Cool programming language into a smart contract version of Cool, which will be referred to BCool programming language.

[000109] BCool

The syntax of an exemplary new programming language, BCool, is identical to the syntax of Cool. Before we define the semantics of BCool, however we fix some notation. Let *tx* be a transaction with inputs *[in_1, ..., in_n]* and outputs *[out_1, ..., out_m]*. An output *out_i* for which *$n < i <= m$* is called a *b-location*. If *$i < min(n, m)$* then out_i is the *successor* of the output *out* that is spent by in_i. In this case, *out* is the *predecessor* of *out_i*. An output *out* is a *revision* if its predecessor is a b-location or a revision. Output *out* is the *latest revision* of output *out'* if there are outputs *out_1, ..., out_n* such that *out' = out_1*, *out_i* is the predecessor of *out_{i+1}* for all *$1 <= i < n$*, *out_n = out*, and *out* is unspent.

[000110] We define two functions that compute the revision and genesis for a given output.

rev(out, B) = out if out is unspent

rev(out, B) = rev(out', B) if out has a successor out'

rev(out, B) = undefined otherwise

loc(out, B) = out if out is a b-location

loc(out, B) = loc(out', B) if out has a predecessor out'

loc(out, B) = undefined otherwise

[000111] This process is shown in Figure 3. Specifically figure 3 is a diagram of the transactions that are broadcast when a location in the store of the decentralized computer is updated.

[000112] Values in BCool, analogous to Cool, are either of a basic type (String, Number, Boolean), void, or of the form $C(a_1 = l_1, \ldots, a_n = l_n)$ where C is a class name, $a_1, \ldots, a_n$ are the attributes of C and $l_1, \ldots, l_n$ are b-locations.

[000113] SEMANTICS OF BCOOL

Let B be a blockchain. A *blockchain environment E for B* is a mapping from identifiers to b-locations in B. The semantic rules of BCool are of the form

so, B, E ⊨ e: v, B'

where *so* is a value, B and B' are blockchains such that $B \subset B'$, E is a blockchain environment, e is an expression, and v is a value with b-locations in B.

[000114] One key distinguishing feature of the present invention, reflected in the exemplary language, BCool, is that it uses a blockchain instead of a store. Whenever an expression e is evaluated, a (possibly empty) set T_e of transactions is broadcast to the blockchain. The BCool semantics uses distinguished outputs called b-locations that correspond to the locations in the Cool semantics. When a location *loc* is updated in Cool, the BCool semantics broadcast a transaction that spends the latest revision of the corresponding b-location into a successor that stores the new value.

[000115] We fix some notations for denoting reads from and writes to a blockchain. Let B be a blockchain and out an output in B. We define B(out) to be the value stored at the latest revision of output out, that is B(out) = v if rev(out) is of the form { data: { val: v }} and B(out) is undefined otherwise.

[000116] Writes are facilitated as follows. Let B be a blockchain, $R = \{ l_1, \ldots, l_n \}$ a set of b-locations in B, and $U = \{ l'_1 \rightarrow v_1, \ldots, l'_m \rightarrow v_m \}$ a mapping from b-locations in B to values such that $\{ l_1, \ldots, l_n \}$ and $\{ l'_1, \ldots, l'_m \}$ are disjoint. We

define *tx(B,U,R)* to be a transaction that spends each location l'_i with $1 <= i <= m$ into a successor that stores the value v_i; tx(B, U, R) also spends the latest revision of each b-location l_i with $1 <= i <= n$ into a successor that stores the value stored at b-location l_i in B:

```
tx(B,U,R) = {
 ins: [
  { old: rev(l1, B) }, ..., { old: rev(ln, B) },
  { old: rev(l'1, B) }, ..., { old: rev(l'm, B) }
 ]
 outs: [
  { data: { val: B(l1) } }, ..., { data: { val: B(ln) } },
  { data: { val: v1 } }, ..., { data: { val: vm } },
 ]
}
```

[000117] Given *tx(B, U, R)*, an expression *e*, we define *tx(B, U, R, e)* to be the transactions obtained from *tx(B, U, R)* by adding expression *e* to the data of the first output.

[000118] Given blockchain *B*, we define *b-newloc(B)* to be a function that returns an unspent b-location from *B*. A unspent b-location can be generated by broadcasting a transaction with more outputs than inputs, so we assume that B contains sufficiently many unspent b-locations to perform a computation.

[000119] Given a blockchain environment $E = [a_1: l_1, ..., a_n: l_n]$ we denote by *loc(E)* the set of b-locations $\{ l_1, ..., l_n \}$ that occur in *E*. Similarly, given value $v=(a_1 = l_1, ..., a_n: l_n)$ we denote by *loc(E)* the set $\{ l_1, ..., l_n \}$

[000120] SEMANTIC RULES

[000121] The semantic rules of BCool can be obtained from the semantic rules of Cool in a mechanical way. BCool has one rule [B-R] that is obtained from the rule [R] of Cool as follows:

- Replace every symboll $\vdash$ for semantic relation of Cool with $\models$ for semantic relation of BCool

- Replace every symbol Si denoting a centralized storage with a symbol Bi denoting a decentralized storage (blockchain)
- Either before or after the store-update replacing step below, ensure that store updates always immediately precede recursive calls or are at the end of a rule body
- Replace every store update $S_i[v_1/l_1, ..., v_n/l_n]$ in the body of a rule for $so, S, E \vdash e: v, S'$ by a blockchain update that adds a transaction $\{ tx(B_i, [v_1/l_1, ..., v_n/l_n], loc(E) \cup loc(so) \setminus \{ l_1, ..., l_n \}, e \}$ to the blockchain Bi. In other words, replace every store update $S_i[v/l]$ that updates at least one location l in S_i by value v, by the instruction to broadcast one or more transactions to the blockchain that spend at least one unspent-transaction-output ("utxo") corresponding to location l into a new output that stores a value v (or stores other meta information that allow v to be computed) and spend at least one additional utxo corresponding to a location in the current execution environment E (also referred to as stack frames) into a new output that stores the same value as its predecessor. Note, the first to bullets merely rewrite the symbols for notation convenience, as the symbol B_i in the added transaction could be written as S_i.

[000122] The rewriting is trivial for rules that do not change the store. Therefore we omit the BCool rules that correspond (letter for letter) to the following Cool rules that do not change the store: [Var], [self], [True], [False], [Int], [String], [If-True], [If-False], [Sequence], [Loop-True], [Loop-False], [IsVoid-True], [IsVoid-False], [Not], [Comp], [Neg], [Arith].

[000123] The rules of the Cool semantics that change the store are [Assign], [Let], [New], [Dispatch], and [StaticDispatch]. Below we explain the rewritings of the first four rules in detail. We omit the rewriting of the rule [StaticDispatch] because it is almost exactly like the rule [Dispatch].

[000124] B-Assign

The semantic rule for variable assignment in BCool is as follows:

$so, B_1, E \models e: v, B_2$

$E(Id) = l$

$B_3 = B_2 \cup \{ tx(B_2, [v/l], loc(E) \cup loc(so) \setminus \{ l \}, Id \leftarrow e) \}$

$so, B_1, E \vDash Id \leftarrow e: v, B_3$

Figure 4 shows the evaluation of Id←e with respect to object so, blockchain B_1, and environment E. In the figure, innermost boxes are either outputs or locations. Outputs are designated as out_i, out'_i, or out''_i for some number i. Locations are designated with the letter l. The top most box labelled B1 represents a set of transactions containing the outputs inside the box labelled B1. Assume that $l_1, \ldots, l_n, l$ are the b-locations that occur in either E or so, and that $out_1, \ldots, out_n$, out are the revisions of $l_1, \ldots, l_n, l$ in B_1 respectively. Like in the semantics of Cool, the first step in the evaluation of *Id ← e* is to recursively evaluate e. Figure 4 shows the set of transactions in T_e that is broadcast when e is evaluated. Transactions in T_e spend the outputs $out_1, \ldots, out_n$, out, thereby creating new revisions $out'_1, \ldots, out'_n$, out'. Then the transaction $tx = tx(B_2, [v/l], loc(E) \cup loc(so) \setminus \{ l \}, Id \leftarrow e)$ is broadcast. This transaction spends the b-locations $loc(E) \cup loc(so) \setminus \{ l \} = \{ l_1, \ldots, l_n \}$ into outputs that store the same values as their predecessor. Transaction tx has one additional input-output pair that spends output out' into an output out" which in turn stores the value v obtained from evaluating e. The first output of tx also stores the expression Id ← e.

[000125] B-Let

the semantic rule for defining variables via *let* is as follows:

$so, B_1, E \vDash e: v, B_2$

$l = b\text{-}newloc(S2)$

$E' = E[l/Id]$

$B_3 = B_2 \cup \{ tx(B_2, [v/l], loc(E') \cup loc(so), let\ Id: T \leftarrow e\ in\ e') \}$

$so, B_3, E' \vDash e': v', B_4$

$so, B_1, E \vDash let\ Id: T \leftarrow e\ in\ e': v', B_4$

[000126] The evaluation of *let Id: T ← e in e'* with respect to a value so, blockchain B_1, and environment E is shown in Figure 5. Assume that $loc(E) \cup loc(so)$

= l_1, ..., l_n are b-locations in B_1 and that out_1, ..., out_n are their revisions in B_1. When the evaluation starts in the first line of the rule of [B-Let], the expression *e* is evaluated and transactions T_e are broadcast that create new revisions out_1', ..., out'_n for the l_1, ..., l_n.

[000127] Next, a b-location *l* is obtained from $B_2 = B_1 \cup T_e$. A blockchain environment *E'* is defined to be *E[l/Id]*. Then the transaction tx(B_2, [v/l], loc(E') ∪ loc(so), let Id: $T_1 \leftarrow e_1$ in e_2) is broadcast. The transaction spends the latest revisions of l_1, ..., l_n into outputs out''_1, ..., out''_n that store the same values as their predecessors. The transaction has one more input-output pair that spends b-location l into a successor that stores the value v.

[000128] In the last step, *e'* is evaluated in the environment E' that contains l_1, ..., l_n form E but also the new location l. The value *v'* returned from evaluating e' is also returned from the evaluation of *let Id: T ← e in e'*.

[000129] B-New

Next, we consider object creation in BCool.

A_0 = C if A = SELF_TYPE and so = C(...) and A_0 = A otherwise

class(A_0) = (a_1: $A_1 \leftarrow e_1$, ..., a_n: $A_n \leftarrow e_n$)

l_i = b-newloc(B_1), for i = 1, ..., n and each l_i is distinct

v = A_0(a_1 = l_1, ..., a_n = l_n)

$B_2 = B_1 \cup \{ tx(B_1, [D_{A\_1}/l_1, ..., D_{A\_n}/l_n], loc(E) \cup loc(so) \setminus \{ l_1, ..., l_n \}, new A) \}$

$v_1, B_2, [a_1: l_1, ..., a_n: l_n] \models \{a_1 \leftarrow e_1; ...; a_n \leftarrow e_n;\}: v', B_3$

---

so, B_1, E ⊨ new T: v, B_3

[000130] Let *A* be a class identifier such that *class(A) = (a_1: $A_1 \leftarrow e_1$, ..., a_n: $A_n \leftarrow e_n$)*. The evaluation of new A with respect to object so, blockchain B_1, and environment E is visualized in Figure 6. When new A is evaluated, *n* unspent b-locations l_1, ..., l_n are obtained from B_1. Assume that loc(E) ∪ loc(so) \ { l_1, ..., l_n } = { l'_1, ..., l'_m } and that { out'_1, ..., out'_m } are their revisions in B_1. When the transaction

$$tx = tx(B_1, [D_{A\_1}/l_1, ..., D_{A\_n}/l_n], loc(E) \cup loc(so) \setminus \{ l_1, ..., l_n \}, new A)$$

is broadcast, the outputs $out'_1, \ldots, out'_m$ are spent into new outputs that store the same values as their predecessors. The fresh b-locations $l_1, \ldots, l_n$ are spent into outputs that store the defaults $D_{A\_1}, \ldots, D_{A\_n}$ of the types $A_1, \ldots, A_n$ of the attributes of A. Finally, each b-locations l_i with $1 <= i <= n$ is initialized to the value of e_i. First e_1 is evaluated in an environment that contains only the b-locations $l_1, \ldots, l_n$. This returns a value v_1 that is assigned to b-location l_1 as described in the section on [B-Assign]. This process is then repeated for each b-location $l_2, \ldots, l_n$ and expression $e_2, \ldots, e_n$. The value $A(a_1 = l_1, \ldots, a_n = l_n)$ is returned from the evaluation of new A.

[000131] B-Dispatch

Finally, the semantics of method dispatch is defined as follows:

$so, B_1, E \models e1: v_1, B_2$

$so, B_2, E \models e2: v_2, B_3$

. . .

$so, B_n, E \models en: v_n, B_{n+1}$

$so, B_{n+1}, E \models e_0: v_0, B_{n+2}$

$A(a_1 = l_{a\_1}, \ldots, a_m = l_{a\_m}) = v_0$

$(x_1, \ldots, x_n, e_{n+1}) = implementation(A, f)$

$l_{x\_i} = b\text{-}newloc(B_{n+2})$, for i = 1, ..., n and each l_{x_i} is distinct

$B_{n+3} = B_{n+2} \cup \{ tx(B_1, [v_1/l_{x\_1}, \ldots, v_n/l_{x\_n}], loc(E) \cup loc(so), e_0.f(e_1, \ldots, e_n)) \}$

$v_0, B_{n+3}, [a_1: l_{a\_1}, \ldots, a_m: l_{a\_m}, x_1: l_{x\_1}, \ldots, x_n: l_{x\_n}] \models e_{n+1}: v_{n+1}, B_{n+4}$

$so, B_1, E \models e_0.f(e_1, \ldots, e_n): v_{n+1}, B_{n+4}$

Figure 7 shows the evaluation of an expression $e_0.f(e_1, \ldots, e_n)$. As before, we consider the evaluation with respect to a blockchain B_1, environment E, and object so. We assume that $loc(E) \cup loc(so) = \{ l'_1, \ldots, l'_k \}$ are b-locations and that $\{ out'_1, \ldots, out'_k \}$ are their revisions in B_1. First, the expressions $e_1, \ldots, e_n$ are evaluated to values $v_1, \ldots, v_n$ in the environment E. Transactions $T_{e\_1} \cup \ldots \cup T_{e\_n}$ are broadcast which updates the blockchain to $B_{n+1} = B_1 \cup T_{e\_1} \cup \ldots \cup T_{e\_n}$. Then, expression e_0 is evaluated in B_{n+1} which returns a value $v_0 = A(a_1 = l_{a\_1}, \ldots, a_m = l_{a\_m})$ and might broadcast further transactions T_{e_0}.

Next, n unspent b-locations $l_{x\_1}, \ldots, l_{x\_n}$ are obtained from $B_{n+2} = B_{n+1} \cup T_{e\_0}$. Then the transaction

$$tx = tx(B_1, [v_1/l_{x\_1}, \ldots, v_n/l_{x\_n}], loc(E) \cup loc(so), e_0.f(e_1, \ldots, e_n))$$

is broadcast. It spends the latest revisions of loc(E) ∪ loc(so) (that is $l'_1, \ldots, l'_k$) into outputs that store the same value as their predecessor. The transaction also spends the unspent b-locations $l_{x\_1}, \ldots, l_{x\_n}$ into revisions that store the values $v_1, \ldots, v_n$. This first output of tx also stores the expression $e_0.f(e_1, \ldots, e_n)$.

Finally, the body e_{n+1} of the function f is evaluated to v_{n+1} in a blockchain environment that contains the b-locations $l_{x\_1}, \ldots, l_{x\_n}$ and $l_{a\_1}, \ldots, l_{a\_m}$. The value v_{n+1} is returned from the evaluation of $e_0.f(e_1, \ldots, e_n)$.

[000132] [B-Static-Dispatch]

The rule for static method dispatch is almost entirely like the rule [B-Dispatch] as shown below. The only difference is that in a normal dispatch, the class of this expression is used to determine the method to invoke; in static dispatch the class is specified in the dispatch itself

$so, B_1, E \vDash e1: v_1, B_2$

$so, B_2, E \vDash e2: v_2, B_3$

. . .

$so, B_n, E \vDash en: v_n, B_{n+1}$

$so, B_{n+1}, E \vDash e_0: v_0, B_{n+2}$

$A(a_1 = l_{a\_1}, \ldots, a_m = l_{a\_m}) = v_0$

$(x_1, \ldots, x_n, e_{n+1}) = implementation(T, f)$

$l_{x\_i} = b\text{-}newloc(B_{n+2})$, for i = 1, …, n and each l_{x_i} is distinct

$B_{n+3} = B_{n+2} \cup \{ tx(B_{n+2}, [v_1/l_{x\_1}, \ldots, v_n/l_{x\_n}], loc(E) \cup loc(so), e_0.f(e_1, \ldots, e_n)) \}$

$v_0, B_{n+3}, [a_1: l_{a\_1}, \ldots, a_m: l_{a\_m}, x_1: l_{x\_1}, \ldots, x_n: l_{x\_n}] \vDash e_{n+1}: v_{n+1}, B_{n+4}$

$so, B_1, E \vDash e_0@T.f(e_1, \ldots, e_n): v_{n+1}, B_{n+4}$

[000133] EXAMPLES:

Following are several examples of the evaluation of BCool expressions.

[000134] Variable Definition and Assignment

Consider the expression **e** below consisting of two nested **let** expressions and an assignment.

```
let x ← 1
 in let y ← 2
  in x ← y
```

The evaluation is shown in Figure 8. The first step is to evaluate the expression 1. This evaluation broadcasts no transactions and returns the value Int(1). Then an unspent b-locations lx is obtained and a transaction is a broadcast that spends lx into an output that stores the value Int(1). The first output of this transaction stores the expression e.

[000135] Then the expression e' = let y←2 in x←y is evaluated in an environment that maps identifier x to location lx. In this evaluation, the first step is to evaluate the expression 2, which returns the value Int(2). A fresh location ly is obtained from the current blockchain. Then a transaction is broadcast that stores e' in its first output. This transaction spends lx because it is in the current environment and spends ly because it is the location created in e'.

[000136] Then the expression e' = let y←2 in x←y is evaluated in an environment that maps identifier x to location lx. In this evaluation, the first step is to evaluate the expression 2, which returns the value Int(2). A fresh location ly is obtained from the current blockchain. Then a transaction is broadcast that stores e' in its first output. This transaction spends lx because it is in the current environment and spends ly because it is the location created in e'.

[000137] In the final step, the assignment x←y is evaluated in the environment E = [x: l_y, y: l_y]. According to the rule for assignments, the expression y is evaluated first. As y is an identifier, no transactions are broadcast. Instead, the b-location l_y for y is looked up in E and the value Int(2) that is stored in the latest revision of l_y is returned. Control returns to the evaluation of x<-y and a transaction is broadcast that spends the latest revisions of the b-locations in the current environment E. The outputs of this transaction update the value stored in l_x to Int(2) and keep all other

locations (only l_y in this case) unchanged. This transaction stores x<-y in its first output.

[000138] Object creation

Consider the program P below:

```
class A {
 num: Int
}
```

Figure 9 shows the evaluation of new A. The first step is to obtain unspent b-locations for each attribute. In this example, there is only one attribute, num, and we denote its unspent b-location by l_{num}. Next, the location l_{num} is initialized to the default value Int(0) for integers. This is achieved by broadcasting a transaction tx that spends l_{num} into an output that stores the value Int(0) and the expression new ANext, consider the class B in the listing below.

```
class B {
  a1: A ← new A
  a2: A ← new A
}
```

[000139] The evaluation of new B is shown in Figure 10. In the first step, two unspent b-locations $la_{\_1}$ and $la_{\_2}$ are obtained from the blockchain. Both are initialized to void in the transaction that also stores the expression new B. The value returned is the object $B(a_1 = la_{\_1}, a_2 = la_{\_2})$.

[000140] Next, the attributes a_1 and a_2 are assigned values. Both assignments are performed in an environment that contains l_{a_1} and l_{a_2}. To assign a value to a_1 a new object of class A is created as described in the last example. As this object is constructed in an environment that contains other locations (l_{a_1} and l_{a_2} in this case), the latest revisions of l_{a_1} and l_{a_2} are spent as well. Next, the value $A(num = l_{num\_1})$ is assigned to the b-location l_{a_1} by the transaction that stores the expression $a_1 \leftarrow$ new A.

[000141] Finally, the second attribute a_2 is initialized in the same way. The return value of the entire expression is $B(a_1 = l_{a\_1}, a_2 = l_{a\_2})$

[000142] Method Invocation

This final example shows how all four rules [B-Let], [B-Assign], [B-New], and [B-Dispatch] work together. Consider the following program:

```
class Counter {
    num: Int
    inc(x: Int): Int {
       num ← x
    }
}
```

The evaluation of the expression e below can be explained as follows

```
let c ← new Counter
in c.inc(1)
```

The first step in the evaluation of e is to evaluate *new Counter*. In the process transactions tx_1 and tx_2 are broadcast as shown in the Figure 11. The value Counter(num = l_{num}) is returned and assigned to the counter variable as follows: a new location l_c is obtained from the blockchain and transaction tx_3 is broadcast that updates the value stored in l_c to Counter(num = l_{num}).

[000143] Finally, c.inc(1) is evaluated in the environment E = [c: l_c]. First, parameter 1 is evaluated to the value Int(1). Then a new unspent b-location l_x is obtained and is initialized to Int(1) by transaction tx_4. Finally, the body *num* ← *x* of the *inc* function is evaluated in the environment E = [num: l_{num}, x: l_x]. This broadcasts the transaction tx_5 that updates the value stored in l_{num} to Int(1). The value returned from the entire expression e is Int(1).

[000144] Completeness of BCool

We associate a store S_B with every blockchain *B* as follows. Let *L* be the set of b-locations in *B* and let Val_L be the set of Cool values with locations in *L*. The *store of B* is a mapping S_B: $L \rightarrow Val_L$ such that $S_B(l)$ is the value stored at the latest revision of $l \in L$.

[000145] **Lemma 1.** Let B be a blockchain with b-locations L. Let $so \in Val_L$, blockchain environment E, blockchain B, and expression e be given and let $S = S_B$. Then so, S, E ⊢ e: v, S' and so, B, E ⊨ s: v, B' implies that S' = $S_{B'}$.

[000146] *Proof.* By induction on e. If e does not change the store in the Cool semantics, then e does not change the blockchain under the BCool semantics and hence the statement is trivial.

[000147] Let so, E, B, and S be as in the assumption. Let $v_1, \ldots, v_n$ be values and let $l_1, \ldots, l_n$ be locations. Let S' = S[v_1/l_1 , ..., v_n/l_n], tx = tx(B, [v_1/l_1 , …, v_n/l_n], loc(E) ∪ loc(so) \ { $l_1, \ldots, l_n$ }, e), and B' = B ∪ { tx }. We show that S' = $S_{B'}$. The Lemma follows by a straightforward induction over e.

[000148] There are two cases. If l = l_i for some 1 <= l <= n, then by definition S'(l) = v_i. Note that tx(B_i, [v_1/l_1 , …, v_n/l_n], loc(E) ∪ loc(so) \ { $l_1, \ldots, l_n$ }, e) spends the latest revision of l into a successor labelled v_i. Hence B'(l) = v_i.

[000149] If l ∉ { $l_1, \ldots, l_n$ } then S'(l) = S(l). If l ∈ loc(E) ∪ loc(so) then tx spends the latest revision of l into a successor that stores the value B(l), and hence B'(l) = B(l) = S(l) = S'(l). Otherwise tx does not spend the latest revision of l and the result follows trivially.

[000150] BCool semantics can be optimized further by, for example, using the following two techniques. The goal of the optimizations is to save costs when running a smart contract. To goal of the first optimization is to reduce the size of each transaction broadcast by not storing the values. The goal of the second optimization is to reduce the number of transactions broadcast. The result of applying both optimizations is a smart contract system that can run programs of arbitrary computational (time and space) complexity at a fixed cost.

[000151] DECREASING SIZE OF TRANSACTIONS: We define an alternative semantics for BCool expressions called the BC2 that broadcasts transactions that store only expressions and no values. The values, nevertheless, can be computed from the

expressions stored on the blockchain. Below, we will refer to the BCool semantics defined above as the BC1. We define a new semantic relation $\models_2$ for BC2 that has one semantic rule [B2-R] for every rule [B-R] in BC1. The rule [B2-R] is obtained from [B-R] by replacing every occurrence of $\models$ by $\models_2$ and every line of the form $B' = B \cup \{ tx(B, [v_1/l_1, \ldots, v_n/l_n], loc(E) \cup loc(so), e) \}$ by a line of the form $B' = B \cup \{ tx(B, [], \{ l_1, \ldots, l_n \} \cup loc(E) \cup loc(so), e) \}$

[000152] Note that the BC2 stores only expressions in the blockchain and no values. Therefore, using BC2 has the advantage that the size of a transaction is reduced and hence transaction fees are saved.

[000153] We define the notion of a valid set of transactions for BC2. Valid transactions are built up from patterns called contexts. A context consists of a sequence of *c-transactions* and *holes*. Every context contains one distinguished c-transaction called the *root* of the context. Every hole contains a list of *positions* and every c-transaction contains a list of *c-outputs*. Holes and c-transactions are labeled by an expression. A node is either a position or an output and there is a binary relation called *spend* on nodes. Figure 12 is a diagram of a transaction context in accordance with some embodiments of the present invention.

[000154] Figures 15 through 18 show four contexts labelled [Ctx-Assign], [Ctx-Let], [Ctx-New], [Ctx-Dispatch], respectively, the relation spend is indicated by the arrows. Note that the BC2 stores only expressions in the blockchain and no values. Therefore, using BC2 has the advantage that the size of a transaction is reduced, and hence transaction fees are saved. Nested boxes with solid lines represent transactions and outputs and arrows represent spending relations. Rectangles with a dashed border represent *holes* that contain *positions*. Each context has one transaction with a thicker border that is called the root of the context. In each context, a set of outputs is shown in backslash crosshatches. The locations of these outputs are called the *connectors* of the context.

[000155] Figure 13 shows context C with *n* connectors. This context can be plugged into a hole *h* of a context C' with *n* positions if the hole and the context are labeled by the same expression. We denote the resulting context by C'[h<-C]. The root of C'[h<-C] is the root of C'.

[000156] An empty context can be plugged into any hole. This process is shown in Figure 14.

[000157] A transaction pattern is a context that contains only nodes. The transaction pattern for expression e is obtained recursively by plugging the transaction pattern for each subexpression e' of e into the e' labeled hole of the context for e. A set of transactions T satisfies a transaction pattern P if there is a one to one mapping iso: T -> P between the c-transactions in P and the transactions in T, such that

- t in T has n inputs and m outputs if and only if iso(t) has n c-outputs with in-degree >0 and m c-outputs with out-degree >0, and
- the i-th input of t spends by the j-th out of t' if and only if there is an edge from iso(t) to iso(t')
- the expression on the first inputs of t is the expression on the label of iso(t).
- A set of transactions T is valid if it satisfies some transaction pattern that can be obtained from the contexts shown in Figures 15-18.

[000158] The following lemma allows multiple users to read the expressions in transactions that are broadcast by BC2 and all perform the same computations on their own devices using the Cool semantics. This way, users can gain consensus over the state of a system that they can modify according to predefined rules.

[000159] **Lemma 2**: For every blockchain B, there is a store S_B such that if T is a valid set of transactions then one can compute a value $so_{T,B}$, an environment $E_{B,T}$, and an expression $e_{B,T}$, such that $so_{B,T}$, S_B, $E_{B,T} \vdash e_{B,T}$: v, S' for some v and $S' = S_{B \cup T}$.

[000160] Proof. Let B and T be given. The proof is by induction on the size of $B \cup T$. As T is valid it satisfies a transaction pattern $P = [ct_1, \ldots, ct_n]$ obtained from the contexts in Figures 15-18. Then P can be parsed into one of the contexts $[h_1, \ldots, h_n, r, h'_1, \ldots, h'_m]$ of Figures 15-18 where r is the root of P.

[000161] Let T_1,..., T_n, be the subsets of T that correspond to the holes h_1, …, h_n of the pattern P under the mapping iso from T to P. Let $B_i = B \cup T_1 \cup \ldots \cup T_i$, for 1 <= i <=n. We apply the induction hypothesis sequentially to B_1 …, B_n. By induction, we compute a values so_i, environments E_i, and expressions e_i such that so_i, S_i, E_i |- e_i: v_i, S_{i+1} where $S_i = S_{B\_i}$ for 1 <= i <= n..

[000162] Next, given S_n, we computer the store S_{n+1} for the blockchain $B_n \cup \{r\}$. Let e_0 be the expression on the first output of r.

[000163] If the e_0 is of the form Id<-e, then n=1 and we lookup Id in the environment E_1 to determine its location l. We define $S_3 = S_2[v_1/l]$, where v_1 is the value we computed from the transactions for e in step 1.

[000164] We next define the self-object so_{n+1}. We first determine the locations of so_{n+1}. Let $Outs_{so}$ be the sequence outputs of r that have the self-object bit set. For each output out in $Outs_{so}$ let l_{out} be its location. We next determine an identifier Id_{out} for each out $\in$ $Outs_{so}$. Consider the expression e_{gen} stored in the transaction t_{gen} that spends l_{out}. One can determine from Figures 15-18 that e_{gen} must be a let-expression, a new-expression, or a dispatch-expression. If e_{gen} = Id<-e, then we define Id_{out} = Id. If e_{gen} is new A such that class(A) = (a_1: T_1 <- e_1, …, a_i: T_i <- e_i) and out is the j-th output of r, then $Id_{out} = a_j$. If $e_{gen} = e'_0.f(e'_1, \ldots, e'_n)$, A is the type of e'_0, implementation(A, f) = (x_1, …, x_n, e_{body}), and out is the j-th output of r, then we define $Id_{out} = x_j$. We define so_{i+1} to be (Id_{out_1}=l_{out_1}, …, Id_{out_k}=l_{out_k}) such that $Outs_{so}$ = [out_1, …, out_k]. The environment E_{n+1} can be constructed from the outputs $Outs_E$ of r that have the environment bit set in the same way. Let e_{n+1} be the expression e_0 = Id<-e stored in the first output of r. One can check that so_{n+1}, S_{n+1}, E_{n+1} |- e_{n+1}: v_{n+1}, S_{n+2} for some v_{n+1} and S_{n+2}.

[000165] The case where e_0 is of the for let Id<-e in e' is similar: Again, by inspecting the patters [Ctx-Let], one can verify that so_1, S_1, E_1 were computed in step 1 such that so_1, S_1, E_1 |- e_1: v_1, S_2. Also, we can tell that r must have k'+1 outputs and k' inputs for some number k'. Let out be the last output of r and let l_{out} be its location. We define $S_3 = S_2[v_1\backslash l_{out}]$ and e_{n+1} to be let Id<-e in e'. Again it is easy to check that so_{n+1}, S_{n+1}, E_{n+1} |- e_{n+1}: v_{n+1}, S_{n+2} for some v_{n+1} and S_{n+2}.

[000166] The cases for e_0 = new A is based on the same ideas. The object so and the environment can be obtained as in the cases for assignment and let expressions. We apply the induction hypothesis and update the store locations l_{a_1}, ..., l_{a_n} with the default values D_{T_1}, ..., D_{T_n} for their types t_1, ..., t_n as indicated in Figure 14.

[000167] The case of a method dispatch $e_0 = e.f(e_1, \ldots, e_n)$ is also similar. The value so and the environment E are determined as in the cases before. We apply the induction hypothesis and update the store locations l_{x_1}, ..., l_{x_n} with the default values v_1, ..., v_n computed in the recursive calls as above.

[000168] In the third step we apply the induction hypothesis to the transactions $T'_1, \ldots, T'_m$ that correspond to the holes $h'_1, \ldots, h'_m$ after r. The lemma follows for the last self-object, environment, and store computed.

[000169] Decreasing the Number of Transactions: We can sketch a third semantics BC3 for BCool expressions. The idea is to evaluate the expression in the Cool semantics first and only store one transaction for each expression that is broadcast. However, as expressions can evaluate sub-expressions in a new environment, one may not get away with only spending the environment in each transaction. It is instead necessary to spend every position that was read, while evaluating the expressions e. This makes it possible to compute values instead of storing them as described in the section on BC2.

[000170] The invention has been disclosed in at least three three smart contract semantics for existing object-oriented programming language. The semantics BC2 and BC3 are more practical than BC1 in some applications.

[000171] BC2 makes it very hard for a user to cheat. All a user can do is broadcast a transaction with an invalid expression. In this case, the evaluation just throws an error. Under BC2 it is impossible for a user to broadcast a transaction leads to a state in the store that is not reachable in the program.

[000172] BC3 has practical relevance because of the low cost that it permits users to run a smart contract. This makes it possible to run smart contracts of arbitrary complexity at a fixed cost.

[000173] While the foregoing descriptions disclose specific examples, other examples may be used to achieve similar results. Further, the various features of the foregoing embodiments may be selected and combined to produce numerous variations of improved systems.

[000174] In the foregoing specification, specific embodiments have been described. However, one of ordinary skill in the art appreciates that various modifications and changes can be made without departing from the scope of the invention as set forth in the claims below. Accordingly, the specification and figures are to be regarded in an illustrative rather than a restrictive sense, and all such modifications are intended to be included within the scope of present teachings.

[000175] Moreover, in this document, relational terms such as first and second, and the like may be used solely to distinguish one entity or action from another entity or action without necessarily requiring or implying any actual such relationship or order between such entities or actions. The terms "comprises," "comprising," "has", "having," "includes", "including," "contains", "containing" or any other variation thereof, are intended to cover a non-exclusive inclusion, such that a process, method, article, or apparatus that comprises, has, includes, contains a list of elements does not include only those elements but may include other elements not expressly listed or inherent to such process, method, article, or apparatus. An element proceeded by "comprises ... a", "has ... a", "includes ... a", "contains ... a" does not, without more constraints, preclude the existence of additional identical elements in the process, method, article, or apparatus that comprises, has, includes, contains the element. The terms "a" and "an" are defined as one or more unless explicitly stated otherwise herein. The terms "substantially", "essentially", "approximately", "about" or any other version thereof, are defined as being close to as understood by one of ordinary skill in the art. The term "coupled" as used herein is defined as connected, although not necessarily directly and not necessarily mechanically. A device or structure that is "configured" in a certain way is configured in at least that way but may also be configured in ways that are not listed.

[000176] The Abstract of the Disclosure is provided to allow the reader to quickly ascertain the nature of the technical disclosure. It is submitted with the understanding that it will not be used to interpret or limit the scope or meaning of the claims. In addition, in the foregoing Detailed Description, it can be seen that various features are grouped together in various embodiments for the purpose of streamlining the disclosure. This method of disclosure is not to be interpreted as reflecting an intention that the claimed embodiments require more features than are expressly recited in each claim. Rather, as the following claims reflect, inventive subject matter lies in less than all features of a single disclosed embodiment. Thus, the following claims are hereby incorporated into the Detailed Description, with each claim standing on its own as a separately claimed subject matter.

APPENDIX

The Cool Reference Manual*

Contents

1 Introduction

This manual describes the programming language Cool: the *C*lassroom *O*bject-*O*riented *L*anguage. Cool is a small language that can be implemented with reasonable effort in a one semester course. Still, Cool retains many of the features of modern programming languages including objects, static typing, and automatic memory management.

Cool programs are sets of *classes*. A class encapsulates the variables and procedures of a data type. Instances of a class are *objects*. In Cool, classes and types are identified; i.e., every class defines a type. Classes permit programmers to define new types and associated procedures (or *methods*) specific to those types. Inheritance allows new types to extend the behavior of existing types.

Cool is an *expression* language. Most Cool constructs are expressions, and every expression has a value and a type. Cool is *type safe*: procedures are guaranteed to be applied to data of the correct type. While static typing imposes a strong discipline on programming in Cool, it guarantees that no runtime type errors can arise in the execution of Cool programs.

This manual is divided into informal and formal components. For a short, informal overview, the first half (through Section 9) suffices. The formal description begins with Section 10.

2 Getting Started

The reader who wants to get a sense for Cool at the outset should begin by reading and running the example programs in the directory `~cs164/examples`. Cool source files have extension `.cl` and Cool assembly files have extension `.s`. The Cool compiler is `~cs164/bin/coolc`. To compile a program:

```
coolc [ -o fileout ] file1.cl file2.cl ... filen.cl
```

The compiler compiles the files `file1.cl` through `filen.cl` as if they were concatenated together. Each file must define a set of complete classes—class definitions may not be split across files. The `-o` option specifies an optional name to use for the output assembly code. If `fileout` is not supplied, the output assembly is named `file1.s`.

The `coolc` compiler generates MIPS assembly code. Because not all of the machines the course is using are MIPS-based, Cool programs are run on a MIPS simulator called `spim`. To run a cool program, type

```
% spim
(spim) load "file.s"
(spim) run
```

To run a different program during the same `spim` session, it is necessary to reinitialize the state of the simulator before loading the new assembly file:

```
(spim) reinit
```

An alternative—and faster—way to invoke `spim` is with a file:

```
spim -file file.s
```

This form loads the file, runs the program, and exits `spim` when the program terminates. Be sure that `spim` is invoked using the script `~cs164/bin/spim`. There may be another version of `spim` installed in on some systems, but it will not execute Cool programs. An easy way to be sure of getting the correct

version is to alias `spim` to `~cs164/bin/spim`. The `spim` manual is available on the course Web page and in the course reader.

The following is a complete transcript of the compilation and execution of `~cs164/examples/list.cl`. This program is very silly, but it does serve to illustrate many of the features of Cool.

```
% coolc list.cl
% spim -file list.s
SPIM Version 5.6 of January 18, 1995
Copyright 1990-1994 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README a full copyright notice.
Loaded: /home/ee/cs164/lib/trap.handler
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
COOL program successfully executed
%
```

3 Classes

All code in Cool is organized into classes. Each class definition must be contained in a single source file, but multiple classes may be defined in the same file. Class definitions have the form:

```
class <type> [ inherits <type> ] {
    <feature_list>
};
```

The notation `[ ...]` denotes an optional construct. All class names are globally visible. Class names begin with an uppercase letter. Classes may not be redefined.

3.1 Features

The body of a class definition consists of a list of feature definitions. A feature is either an *attribute* or a *method*. An attribute of class `A` specifies a variable that is part of the state of objects of class `A`. A method of class `A` is a procedure that may manipulate the variables and objects of class `A`.

One of the major themes of modern programming languages is *information hiding*, which is the idea that certain aspects of a data type's implementation should be abstract and hidden from users of the data type. Cool supports information hiding through a simple mechanism: all attributes have scope local to the class, and all methods have global scope. Thus, the only way to provide access to object state in Cool is through methods.

Feature names must begin with a lowercase letter. No method name may be defined multiple times in a class, and no attribute name may be defined multiple times in a class, but a method and an attribute may have the same name.

A fragment from `list.cl` illustrates simple cases of both attributes and methods:

```
class Cons inherits List {
xcar : Int;
xcdr : List;

isNil() : Bool { false };

init(hd : Int, tl : List) : Cons {
  {
    xcar <- hd;
    xcdr <- tl;
    self;
  }
}
...
};
```

In this example, the class `Cons` has two attributes `xcar` and `xcdr` and two methods `isNil` and `init`. Note that the types of attributes, as well as the types of formal parameters and return types of methods, are explicitly declared by the programmer.

Given object `c` of class `Cons` and object `l` of class `List`, we can set the `xcar` and `xcdr` fields by using the method `init`:

```
c.init(1,l)
```

This notation is *object-oriented dispatch*. There may be many definitions of `init` methods in many different classes. The dispatch looks up the class of the object `c` to decide which `init` method to invoke. Because the class of `c` is `Cons`, the `init` method in the `Cons` class is invoked. Within the invocation, the variables `xcar` and `xcdr` refer to `c`'s attributes. The special variable `self` refers to the object on which the method was dispatched, which, in the example, is `c` itself.

There is a special form `new C` that generates a fresh object of class `C`. An object can be thought of as a record that has a slot for each of the attributes of the class as well as pointers to the methods of the class. A typical dispatch for the `init` method is:

```
(new Cons).init(1,new Nil)
```

This example creates a new cons cell and initializes the "car" of the cons cell to be `1` and the "cdr" to be `new Nil`.[1] There is no mechanism in Cool for programmers to deallocate objects. Cool has *automatic memory management*; objects that cannot be used by the program are deallocated by a runtime garbage collector.

Attributes are discussed further in Section 5 and methods are discussed further in Section 6.

3.2 Inheritance

If a class definition has the form

```
class C inherits P { ... };
```

[1] In this example, `Nil` is assumed to be a subtype of `List`.

then class C inherits the features of P. In this case P is the *parent* class of C and C is a *child* class of P.

The semantics of C inherits P is that C has all of the features defined in P in addition to its own features. In the case that a parent and child both define the same method name, then the definition given in the child class takes precedence. It is illegal to redefine attribute names. Furthermore, for type safety, it is necessary to place some restrictions on how methods may be redefined (see Section 6).

There is a distinguished class Object. If a class definition does not specify a parent class, then the class inherits from Object by default. A class may inherit only from a single class; this is aptly called "single inheritance."[2] The parent-child relation on classes defines a graph. This graph may not contain cycles. For example, if C inherits from P, then P must not inherit from C. Furthermore, if C inherits from P, then P must have a class definition somewhere in the program. Because Cool has single inheritance, it follows that if both of these restrictions are satisfied, then the inheritance graph forms a tree with Object as the root.

In addition to Object, Cool has four other *basic classes*: Int, String, Bool, and IO. The basic classes are discussed in Section 8.

4 Types

In Cool, every class name is also a type. In addition, there is a type SELF_TYPE that can be used in special circumstances.

A *type declaration* has the form x:C, where x is a variable and C is a type. Every variable must have a type declaration at the point it is introduced, whether that is in a let, case, or as the formal parameter of a method. The types of all attributes must also be declared.

The basic type rule in Cool is that if a method or variable expects a value of type P, then any value of type C may be used instead, provided that P is an ancestor of C in the class hierarchy. In other words, if C inherits from P, either directly or indirectly, then a C can be used wherever a P would suffice.

When an object of class C may be used in place of an object of class P, we say that C *conforms* to P or that $C \leq P$ (think: C is lower down in the inheritance tree). As discussed above, conformance is defined in terms of the inheritance graph.

Definition 4.1 (Conformance) Let A, C, and P be types.

- $A \leq A$ for all types A
- if C inherits from P, then $C \leq P$
- if $A \leq C$ and $C \leq P$ then $A \leq P$

Because Object is the root of the class hierarchy, it follows that $A \leq$ Object for all types A.

4.1 SELF_TYPE

The type SELF_TYPE is used to refer to the type of the self variable. This is useful in classes that will be inherited by other classes, because it allows the programmer to avoid specifying a fixed final type at the time the class is written. For example, the program

[2]Some object-oriented languages allow a class to inherit from multiple classes, which is equally aptly called "multiple inheritance."

```
class Silly {
   copy() : SELF_TYPE { self };
};

class Sally inherits Silly { };

class Main {
   x : Sally <- (new Sally).copy();

   main() : Sally { x };
};
```

Because SELF_TYPE is used in the definition of the copy method, we know that the result of copy is the same as the type of the self parameter. Thus, it follows that (new Sally).copy() has type Sally, which conforms to the declaration of attribute x.

Note that the meaning of SELF_TYPE is not fixed, but depends on the class in which it is used. In general, SELF_TYPE may refer to the class C in which it appears, or any class that conforms to C. When it is useful to make explicit what SELF_TYPE may refer to, we use the name of the class C in which SELF_TYPE appears as an index SELF_TYPE_C. This subscript notation is not part of Cool syntax—it is used merely to make clear in what class a particular occurrence of SELF_TYPE appears.

From Definition 4.1, it follows that $\text{SELF\_TYPE}_X \leq \text{SELF\_TYPE}_X$. There is also a special conformance rule for SELF_TYPE:

$$\text{SELF\_TYPE}_C \leq P \text{ if } C \leq P$$

Finally, SELF_TYPE may be used in the following places: new SELF_TYPE, as the return type of a method, as the declared type of a let variable, or as the declared type of an attribute. No other uses of SELF_TYPE are permitted.

4.2 Type Checking

The Cool type system guarantees at compile time that execution of a program cannot result in runtime type errors. Using the type declarations for identifiers supplied by the programmer, the type checker infers a type for every expression in the program.

It is important to distinguish between the type assigned by the type checker to an expression at compile time, which we shall call the *static* type of the expression, and the type(s) to which the expression may evaluate during execution, which we shall call the *dynamic* types.

The distinction between static and dynamic types is needed because the type checker cannot, at compile time, have perfect information about what values will be computed at runtime. Thus, in general, the static and dynamic types may be different. What we require, however, is that the type checker's static types be *sound* with respect to the dynamic types.

Definition 4.2 For any expression e, let D_e be a dynamic type of e and let S_e be the static type inferred by the type checker. Then the type checker is *sound* if for all expressions e it is the case that $D_e \leq S_e$.

Put another way, we require that the type checker err on the side of overestimating the type of an expression in those cases where perfect accuracy is not possible. Such a type checker will never accept a program that contains type errors. However, the price paid is that the type checker will reject some programs that would actually execute without runtime errors.

5 Attributes

An attribute definition has the form

```
<id> : <type> [ <- <expr> ];
```

The expression is optional initialization that is executed when a new object is created. The static type of the expression must conform to the declared type of the attribute. If no initialization is supplied, then the default initialization is used (see below).

When a new object of a class is created, all of the inherited and local attributes must be initialized. Inherited attributes are initialized first in inheritance order beginning with the attributes of the greatest ancestor class. Within a given class, attributes are initialized in the order they appear in the source text.

Attributes are local to the class in which they are defined or inherited. Inherited attributes cannot be redefined.

5.1 Void

All variables in Cool are initialized to contain values of the appropriate type. The special value `void` is a member of all types and is used as the default initialization for variables where no initialization is supplied by the user. (`void` is used where one would use `NULL` in C or `null` in Java; Cool does not have anything equivalent to C's or Java's `void` type.) Note that there is no name for `void` in Cool; the only way to create a `void` value is to declare a variable of some class other than `Int`, `String`, or `Bool` and allow the default initialization to occur, or to store the result of a `while` loop.

There is a special form `isvoid expr` that tests whether a value is `void` (see Section 7.11). In addition, `void` values may be tested for equality. A `void` value may be passed as an argument, assigned to a variable, or otherwise used in any context where any value is legitimate, except that a dispatch to or case on `void` generates a runtime error.

Variables of the basic classes `Int`, `Bool`, and `String` are initialized specially; see Section 8.

6 Methods

A method definition has the form

```
<id>(<id> : <type>,...,<id> : <type>): <type> { <expr> };
```

There may be zero or more formal parameters. The identifiers used in the formal parameter list must be distinct. The type of the method body must conform to the declared return type. When a method is invoked, the formal parameters are bound to the actual arguments and the expression is evaluated; the resulting value is the meaning of the method invocation. A formal parameter hides any definition of an attribute of the same name.

To ensure type safety, there are restrictions on the redefinition of inherited methods. The rule is simple: If a class `C` inherits a method `f` from an ancestor class `P`, then `C` may override the inherited definition of `f` provided the number of arguments, the types of the formal parameters, and the return type are exactly the same in both definitions.

To see why some restriction is necessary on the redefinition of inherited methods, consider the following example:

```
class P {
   f(): Int { 1 };
};

class C inherits P {
   f(): String { "1" };
};
```

Let `p` be an object with dynamic type `P`. Then

```
p.f() + 1
```

is a well-formed expression with value 2. However, we cannot substitute a value of type `C` for `p`, as it would result in adding a string to a number. Thus, if methods can be redefined arbitrarily, then subclasses may not simply extend the behavior of their parents, and much of the usefulness of inheritance, as well as type safety, is lost.

7 Expressions

Expressions are the largest syntactic category in Cool.

7.1 Constants

The simplest expressions are constants. The boolean constants are `true` and `false`. Integer constants are unsigned strings of digits such as `0`, `123`, and `007`. String constants are sequences of characters enclosed in double quotes, such as `"This is a string."` String constants may be at most 1024 characters long. There are other restrictions on strings; see Section 10.

The constants belong to the basic classes `Bool`, `Int`, and `String`. The value of a constant is an object of the appropriate basic class.

7.2 Identifiers

The names of local variables, formal parameters of methods, `self`, and class attributes are all expressions. The identifier `self` may be referenced, but it is an error to assign to `self` or to bind `self` in a `let`, a `case`, or as a formal parameter. It is also illegal to have attributes named `self`.

Local variables and formal parameters have lexical scope. Attributes are visible throughout a class in which they are declared or inherited, although they may be hidden by local declarations within expressions. The binding of an identifier reference is the innermost scope that contains a declaration for that identifier, or to the attribute of the same name if there is no other declaration. The exception to this rule is the identifier `self`, which is implicitly bound in every class.

7.3 Assignment

An assignment has the form

```
<id> <- <expr>
```

The static type of the expression must conform to the declared type of the identifier. The value is the value of the expression. The static type of an assignment is the static type of `<expr>`.

7.4 Dispatch

There are three forms of dispatch (i.e. method call) in Cool. The three forms differ only in how the called method is selected. The most commonly used form of dispatch is

```
<expr>.<id>(<expr>,...,<expr>)
```

Consider the dispatch $\texttt{e}_0\texttt{.f}(\texttt{e}_1,\ldots,\texttt{e}_n)$. To evaluate this expression, the arguments are evaluated in left-to-right order, from $\texttt{e}_1$ to $\texttt{e}_n$. Next, $\texttt{e}_0$ is evaluated and its class `C` noted (if $\texttt{e}_0$ is `void` a runtime error is generated). Finally, the method `f` in class `C` is invoked, with the value of $\texttt{e}_0$ bound to `self` in the body of `f` and the actual arguments bound to the formals as usual. The value of the expression is the value returned by the method invocation.

Type checking a dispatch involves several steps. Assume $\texttt{e}_0$ has static type `A`. (Recall that this type is not necessarily the same as the type `C` above. `A` is the type inferred by the type checker; `C` is the class of the object computed at runtime, which is potentially any subclass of `A`.) Class `A` must have a method `f`, the dispatch and the definition of `f` must have the same number of arguments, and the static type of the ith actual parameter must conform to the declared type of the ith formal parameter.

If `f` has return type `B` and `B` is a class name, then the static type of the dispatch is `B`. Otherwise, if `f` has return type `SELF_TYPE`, then the static type of the dispatch is `A`. To see why this is sound, note that the `self` parameter of the method `f` conforms to type `A`. Therefore, because `f` returns `SELF_TYPE`, we can infer that the result must also conform to `A`. Inferring accurate static types for dispatch expressions is what justifies including `SELF_TYPE` in the Cool type system.

The other forms of dispatch are:

```
<id>(<expr>,...,<expr>)
<expr>@<type>.id(<expr>,...,<expr>)
```

The first form is shorthand for `self.<id>(<expr>,...,<expr>)`.

The second form provides a way of accessing methods of parent classes that have been hidden by redefinitions in child classes. Instead of using the class of the leftmost expression to determine the method, the method of the class explicitly specified is used. For example, `e@B.f()` invokes the method `f` in class `B` on the object that is the value of `e`. For this form of dispatch, the static type to the left of "@"must conform to the type specified to the right of "@".

7.5 Conditionals

A conditional has the form

```
if <expr> then <expr> else <expr> fi
```

The semantics of conditionals is standard. The predicate is evaluated first. If the predicate is `true`, then the `then` branch is evaluated. If the predicate is `false`, then the `else` branch is evaluated. The value of the conditional is the value of the evaluated branch.

The predicate must have static type `Bool`. The branches may have any static types. To specify the static type of the conditional, we define an operation $\sqcup$ (pronounced "join") on types as follows. Let `A,B,D` be any types other than `SELF_TYPE`. The *least type* of a set of types means the least element with respect to the conformance relation $\leq$.

$$
\begin{aligned}
\texttt{A} \sqcup \texttt{B} &= \text{the least type } \texttt{C} \text{ such that } \texttt{A} \leq \texttt{C} \text{ and } \texttt{B} \leq \texttt{C} \\
\texttt{A} \sqcup \texttt{A} &= \texttt{A} \qquad \text{(idempotent)} \\
\texttt{A} \sqcup \texttt{B} &= \texttt{B} \sqcup \texttt{A} \qquad \text{(commutative)} \\
\texttt{SELF\_TYPE}_\texttt{D} \sqcup \texttt{A} &= \texttt{D} \sqcup \texttt{A}
\end{aligned}
$$

Let `T` and `F` be the static types of the branches of the conditional. Then the static type of the conditional is `T` ⊔ `F`. (think: Walk towards `Object` from each of `T` and `F` until the paths meet.)

7.6 Loops

A loop has the form

```
while <expr> loop <expr> pool
```

The predicate is evaluated before each iteration of the loop. If the predicate is `false`, the loop terminates and `void` is returned. If the predicate is `true`, the body of the loop is evaluated and the process repeats.

The predicate must have static type `Bool`. The body may have any static type. The static type of a loop expression is `Object`.

7.7 Blocks

A block has the form

```
{ <expr>; ... <expr>; }
```

The expressions are evaluated in left-to-right order. Every block has at least one expression; the value of a block is the value of the last expression. The expressions of a block may have any static types. The static type of a block is the static type of the last expression.

An occasional source of confusion in Cool is the use of semi-colons (";"). Semi-colons are used as terminators in lists of expressions (e.g., the block syntax above) and not as expression separators. Semi-colons also terminate other Cool constructs, see Section 11 for details.

7.8 Let

A let expression has the form

```
let <id1> : <type1> [ <- <expr1> ], ..., <idn> : <typen> [ <- <exprn> ] in <expr>
```

The optional expressions are *initialization*; the other expression is the *body*. A `let` is evaluated as follows. First `<expr1>` is evaluated and the result bound to `<id1>`. Then `<expr2>` is evaluated and the result bound to `<id2>`, and so on, until all of the variables in the `let` are initialized. (If the initialization of `<idk>` is omitted, the default initialization of type `<typek>` is used.) Next the body of the `let` is evaluated. The value of the `let` is the value of the body.

The `let` identifiers `<id1>,...,<idn>` are visible in the body of the `let`. Furthermore, identifiers `<id1>,...,<idk>` are visible in the initialization of `<idm>` for any `m > k`.

If an identifier is defined multiple times in a `let`, later bindings hide earlier ones. Identifiers introduced by `let` also hide any definitions for the same names in containing scopes. Every `let` expression must introduce at least one identifier.

The type of an initialization expression must conform to the declared type of the identifier. The type of `let` is the type of the body.

The `<expr>` of a `let` extends as far (encompasses as many tokens) as the grammar allows.

7.9 Case

A case expression has the form

```
case <expr0> of
    <id1> : <type1> => <expr1>;
    . . .
    <idn> : <typen> => <exprn>;
esac
```

Case expressions provide runtime type tests on objects. First, `expr0` is evaluated and its dynamic type `C` noted (if `expr0` evaluates to `void` a run-time error is produced). Next, from among the branches the branch with the least type `<typek>` such that $C \leq$ `<typek>` is selected. The identifier `<idk>` is bound to the value of `<expr0>` and the expression `<exprk>` is evaluated. The result of the `case` is the value of `<exprk>`. If no branch can be selected for evaluation, a run-time error is generated. Every `case` expression must have at least one branch.

For each branch, let T_i be the static type of `<expri>`. The static type of a `case` expression is $\bigsqcup_{1 \leq i \leq n} T_i$. The identifier `id` introduced by a branch of a `case` hides any variable or attribute definition for `id` visible in the containing scope.

The `case` expression has no special construct for a "default" or "otherwise" branch. The same affect is achieved by including a branch

```
x : Object => ...
```

because every type is $\leq$ to `Object`.

The `case` expression provides programmers a way to insert explicit runtime type checks in situations where static types inferred by the type checker are too conservative. A typical situation is that a programmer writes an expression `e` and type checking infers that `e` has static type `P`. However, the programmer may know that, in fact, the dynamic type of `e` is always `C` for some $C \leq P$. This information can be captured using a case expression:

```
case e of x : C => ...
```

In the branch the variable `x` is bound to the value of `e` but has the more specific static type `C`.

7.10 New

A `new` expression has the form

```
new <type>
```

The value is a fresh object of the appropriate class. If the type is `SELF_TYPE`, then the value is a fresh object of the class of `self` in the current scope. The static type is `<type>`.

7.11 Isvoid

The expression

```
isvoid expr
```

evaluates to `true` if `expr` is `void` and evaluates to `false` if `expr` is not `void`.

7.12 Arithmetic and Comparison Operations

Cool has four binary arithmetic operations: `+, -, *, /`. The syntax is

```
expr1 <op> expr2
```

To evaluate such an expression first `expr1` is evaluated and then `expr2`. The result of the operation is the result of the expression.

The static types of the two sub-expressions must be `Int`. The static type of the expression is `Int`. Cool has only integer division.

Cool has three comparison operations: `<, <=, =`. For `<` and `<=` the rules are exactly the same as for the binary arithmetic operations, except that the result is a `Bool`. The comparison `=` is a special case. If either `<expr1>` or `<expr2>` has static type `Int`, `Bool`, or `String`, then the other must have the same static type. Any other types, including `SELF_TYPE`, may be freely compared. On non-basic objects, equality simply checks for pointer equality (i.e., whether the memory addresses of the objects are the same). Equality is defined for `void`.

In principle, there is nothing wrong with permitting equality tests between, for example, `Bool` and `Int`. However, such a test must always be false and almost certainly indicates some sort of programming error. The Cool type checking rules catch such errors at compile-time instead of waiting until runtime.

Finally, there is one arithmetic and one logical unary operator. The expression `~<expr>` is the integer complement of `<expr>`. The expression `<expr>` must have static type `Int` and the entire expression has static type `Int`. The expression `not <expr>` is the boolean complement of `<expr>`. The expression `<expr>` must have static type `Bool` and the entire expression has static type `Bool`.

8 Basic Classes

8.1 Object

The `Object` class is the root of the inheritance graph. Methods with the following declarations are defined:

```
abort() : Object
type_name() : String
copy() : SELF_TYPE
```

The method `abort` halts program execution with an error message. The method `type_name` returns a string with the name of the class of the object. The method `copy` produces a *shallow* copy of the object.[3]

8.2 IO

The `IO` class provides the following methods for performing simple input and output operations:

```
out_string(x : String) : SELF_TYPE
out_int(x : Int) : SELF_TYPE
in_string() : String
in_int() : Int
```

[3]A shallow copy of *a* copies *a* itself, but does not recursively copy objects that *a* points to.

The methods `out_string` and `out_int` print their argument and return their `self` parameter. The method `in_string` reads a string from the standard input, up to but not including a newline character. The method `in_int` reads a single integer, which may be preceded by whitespace. Any characters following the integer, up to and including the next newline, are discarded by `in_int`.

A class can make use of the methods in the `IO` class by inheriting from `IO`. It is an error to redefine the `IO` class.

8.3 Int

The `Int` class provides integers. There are no methods special to `Int`. The default initialization for variables of type `Int` is 0 (not `void`). It is an error to inherit from or redefine `Int`.

8.4 String

The `String` class provides strings. The following methods are defined:

```
length() : Int
concat(s : String) : String
substr(i : Int, l : Int) : String
```

The method `length` returns the length of the `self` parameter. The method `concat` returns the string formed by concatenating `s` after `self`. The method `substr` returns the substring of its `self` parameter beginning at position `i` with length `l`; string positions are numbered beginning at 0. A runtime error is generated if the specified substring is out of range.

The default initialization for variables of type `String` is `""` (not `void`). It is an error to inherit from or redefine `String`.

8.5 Bool

The `Bool` class provides `true` and `false`. The default initialization for variables of type `Bool` is `false` (not `void`). It is an error to inherit from or redefine `Bool`.

9 Main Class

Every program must have a class `Main`. Furthermore, the `Main` class must have a method `main` that takes no formal parameters. The `main` method must be defined in class `Main` (not inherited from another class). A program is executed by evaluating `(new Main).main()`.

The remaining sections of this manual provide a more formal definition of Cool. There are four sections covering lexical structure (Section 10), grammar (Section 11), type rules (Section 12), and operational semantics (Section 13).

10 Lexical Structure

The lexical units of Cool are integers, type identifiers, object identifiers, special notation, strings, keywords, and white space.

10.1 Integers, Identifiers, and Special Notation

Integers are non-empty strings of digits 0-9. Identifiers are strings (other than keywords) consisting of letters, digits, and the underscore character. Type identifiers begin with a capital letter; object identifiers begin with a lower case letter. There are two other identifiers, **self** and **SELF_TYPE** that are treated specially by Cool but are not treated as keywords. The special syntactic symbols (e.g., parentheses, assignment operator, etc.) are given in Figure 1.

10.2 Strings

Strings are enclosed in double quotes `"..."`. Within a string, a sequence '\c' denotes the character 'c', with the exception of the following:

\b	backspace
\t	tab
\n	newline
\f	formfeed

A non-escaped newline character may not appear in a string:

```
"This \
is OK"
"This is not
OK"
```

A string may not contain EOF. A string may not contain the null (character \0). Any other character may be included in a string. Strings cannot cross file boundaries.

10.3 Comments

There are two forms of comments in Cool. Any characters between two dashes "--" and the next newline (or EOF, if there is no next newline) are treated as comments. Comments may also be written by enclosing text in $(*\ldots*)$. The latter form of comment may be nested. Comments cannot cross file boundaries.

10.4 Keywords

The keywords of cool are: **class, else, false, fi, if, in, inherits, isvoid, let, loop, pool, then, while, case, esac, new, of, not, true.** Except for the constants **true** and **false**, keywords are case insensitive. To conform to the rules for other objects, the first letter of **true** and **false** must be lowercase; the trailing letters may be upper or lower case.

10.5 White Space

White space consists of any sequence of the characters: blank (ascii 32), `\n` (newline, ascii 10), `\f` (form feed, ascii 12), `\r` (carriage return, ascii 13), `\t` (tab, ascii 9), `\v` (vertical tab, ascii 11).

$$
\begin{array}{rcl}
\textit{program} & ::= & [\![\textit{class};]\!]^{+} \\
\textit{class} & ::= & \textbf{class}\ \text{TYPE}\ [\textbf{inherits}\ \text{TYPE}]\ \{\ [\![\textit{feature};]\!]^{*}\} \\
\textit{feature} & ::= & \text{ID}(\ [\ \textit{formal}\ [\![, \textit{formal}]\!]^{*}\ ]\ ) : \text{TYPE}\ \{\ \textit{expr}\ \} \\
 & | & \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ] \\
\textit{formal} & ::= & \text{ID} : \text{TYPE} \\
\textit{expr} & ::= & \text{ID}\ \texttt{<-}\ \textit{expr} \\
 & | & \textit{expr}[@\text{TYPE}].\text{ID}(\ [\ \textit{expr}\ [\![, \textit{expr}]\!]^{*}\ ]\ ) \\
 & | & \text{ID}(\ [\ \textit{expr}\ [\![, \textit{expr}]\!]^{*}\ ]\ ) \\
 & | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{expr}\ \textbf{else}\ \textit{expr}\ \textbf{fi} \\
 & | & \textbf{while}\ \textit{expr}\ \textbf{loop}\ \textit{expr}\ \textbf{pool} \\
 & | & \{\ [\![\textit{expr};]\!]^{+}\} \\
 & | & \textbf{let}\ \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]\ [\![, \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]]\!]^{*}\ \textbf{in}\ \textit{expr} \\
 & | & \textbf{case}\ \textit{expr}\ \textbf{of}\ [\![\text{ID} : \text{TYPE} \Rightarrow \textit{expr};]\!]^{+}\textbf{esac} \\
 & | & \textbf{new}\ \text{TYPE} \\
 & | & \textbf{isvoid}\ \textit{expr} \\
 & | & \textit{expr} + \textit{expr} \\
 & | & \textit{expr} - \textit{expr} \\
 & | & \textit{expr} * \textit{expr} \\
 & | & \textit{expr} / \textit{expr} \\
 & | & \tilde{}\,\textit{expr} \\
 & | & \textit{expr} < \textit{expr} \\
 & | & \textit{expr} <= \textit{expr} \\
 & | & \textit{expr} = \textit{expr} \\
 & | & \textbf{not}\ \textit{expr} \\
 & | & (\textit{expr}) \\
 & | & \text{ID} \\
 & | & \text{integer} \\
 & | & \text{string} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

Figure 1: Cool syntax.

11 Cool Syntax

Figure 1 provides a specification of Cool syntax. The specification is not in pure Backus-Naur Form (BNF); for convenience, we also use some regular expression notation. Specifically, A^* means zero or more A's in succession; A^+ means one or more A's. Items in square brackets [...] are optional. Double brackets $[\![\ ]\!]$ are not part of Cool; they are used in the grammar as a meta-symbol to show association of grammar symbols (e.g. $a[\![bc]\!]^+$ means a followed by one or more bc pairs).

11.1 Precedence

The precedence of infix binary and prefix unary operations, from highest to lowest, is given by the following table:

```
.
@
~
isvoid
* /
+ -
<=  <  =
not
<-
```

All binary operations are left-associative, with the exception of assignment, which is right-associative, and the three comparison operations, which do not associate.

12 Type Rules

This section formally defines the type rules of Cool. The type rules define the type of every Cool expression in a given context. The context is the *type environment*, which describes the type of every unbound identifier appearing in an expression. The type environment is described in Section 12.1. Section 12.2 gives the type rules.

12.1 Type Environments

To a first approximation, type checking in Cool can be thought of as a bottom-up algorithm: the type of an expression e is computed from the (previously computed) types of e's subexpressions. For example, an integer `1` has type `Int`; there are no subexpressions in this case. As another example, if $\texttt{e}_n$ has type `X`, then the expression $\{\ \texttt{e}_1; \ldots; \texttt{e}_n;\ \}$ has type `X`.

A complication arises in the case of an expression `v`, where `v` is an object identifier. It is not possible to say what the type of `v` is in a strictly bottom-up algorithm; we need to know the type declared for `v` in the larger expression. Such a declaration must exist for every object identifier in valid Cool programs.

To capture information about the types of identifiers, we use a *type environment*. The environment consists of three parts: a method environment M, an object environment O, and the name of the current class in which the expression appears. The method environment and object environment are both functions (also called *mappings*). The object environment is a function of the form

$$O(v) = T$$

which assigns the type T to object identifier v. The method environment is more complex; it is a function of the form

$$M(C, f) = (T_1, \ldots, T_{n-1}, T_n)$$

where C is a class name (a type), f is a method name, and $t_1, \ldots, t_n$ are types. The tuple of types is the *signature* of the method. The interpretation of signatures is that in class C the method f has formal parameters of types $(t_1, \ldots, t_{n-1})$—in that order—and a return type t_n.

Two mappings are required instead of one because object names and method names do not clash—i.e., there may be a method and an object identifier of the same name.

The third component of the type environment is the name of the current class, which is needed for type rules involving `SELF_TYPE`.

Every expression e is type checked in a type environment; the subexpressions of e may be type checked in the same environment or, if e introduces a new object identifier, in a modified environment. For example, consider the expression

```
let c : Int <- 33 in
  ...
```

The `let` expression introduces a new variable `c` with type `Int`. Let O be the object component of the type environment for the `let`. Then the body of the `let` is type checked in the object type environment

$$O[Int/c]$$

where the notation $O[T/c]$ is defined as follows:

$$\begin{aligned} O[T/c](c) &= T \\ O[T/c](d) &= O(d) \quad \text{if } d \neq c \end{aligned}$$

12.2 Type Checking Rules

The general form a type checking rule is:

$$\frac{\vdots}{O, M, C \vdash e : T}$$

The rule should be read: In the type environment for objects O, methods M, and containing class C, the expression e has type T. The dots above the horizontal bar stand for other statements about the types of sub-expressions of e. These other statements are hypotheses of the rule; if the hypotheses are satisfied, then the statement below the bar is true. In the conclusion, the "turnstyle" ("$\vdash$") separates context (O, M, C) from statement $(e : T)$.

The rule for object identifiers is simply that if the environment assigns an identifier Id type T, then Id has type T.

$$\frac{O(Id) = T}{O, M, C \vdash Id : T} \qquad \text{[Var]}$$

The rule for assignment to a variable is more complex:

$$\frac{\begin{array}{l} O(Id) = T \\ O, M, C \vdash e_1 : T' \\ T' \leq T \end{array}}{O, M, C \vdash Id \leftarrow e_1 : T'} \qquad \text{[ASSIGN]}$$

Note that this type rule—as well as others—use the conformance relation $\leq$ (see Section 3.2). The rule says that the assigned expression e_1 must have a type T' that conforms to the type T of the identifier Id in the type environment. The type of the whole expression is T'.

The type rules for constants are all easy:

$$\frac{}{O, M, C \vdash true : Bool} \quad \text{[True]}$$

$$\frac{}{O, M, C \vdash false : Bool} \quad \text{[False]}$$

$$\frac{i \text{ is an integer constant}}{O, M, C \vdash i : Int} \quad \text{[Int]}$$

$$\frac{s \text{ is a string constant}}{O, M, C \vdash s : String} \quad \text{[String]}$$

There are two cases for `new`, one for `new SELF_TYPE` and one for any other form:

$$\frac{T' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T = \texttt{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}}{O, M, C \vdash new\ T : T'} \quad \text{[New]}$$

Dispatch expressions are the most complex to type check.

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0' = \begin{cases} C & \text{if } T_0 = \texttt{SELF\_TYPE}_C \\ T_0 & \text{otherwise} \end{cases} \\ M(T_0', f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[Dispatch]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[StaticDispatch]}$$

To type check a dispatch, each of the subexpressions must first be type checked. The type T_0 of e_0 determines which declaration of the method f is used. The argument types of the dispatch must conform to the declared argument types. Note that the type of the result of the dispatch is either the declared return type or T_0 in the case that the declared return type is `SELF_TYPE`. The only difference in type checking a static dispatch is that the class T of the method f is given in the dispatch, and the type T_0 must conform to T.

The type checking rules for `if` and {-} expressions are straightforward. See Section 7.5 for the definition of the $\sqcup$ operation.

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \\ O, M, C \vdash e_3 : T_3 \end{array}}{O, M, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : T_2 \sqcup T_3} \quad \text{[If]}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ \vdots \\ O, M, C \vdash e_n : T_n \end{array}}{O, M, C \vdash \{\ e_1; e_2; \ldots e_n;\ \} : T_n} \quad \text{[Sequence]}$$

The `let` rule has some interesting aspects.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0' \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \quad \text{[Let-Init]}$$

First, the initialization e_1 is type checked in an environment without a new definition for x. Thus, the variable x cannot be used in e_1 unless it already has a definition in an outer scope. Second, the body e_2 is type checked in the environment O extended with the typing $x : T_0'$. Third, note that the type of x may be `SELF_TYPE`.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O[T_0'/x], M, C \vdash e_1 : T_1 \end{array}}{O, M, C \vdash \text{let } x : T_0 \text{ in } e_1 : T_1} \quad \text{[Let-No-Init]}$$

The rule for `let` with no initialization simply omits the conformance requirement. We give type rules only for a `let` with a single variable. Typing a multiple `let`

$$\text{let } x_1 : T_1 \ [\leftarrow e_1],\ x_2 : T_2 \ [\leftarrow e2], \ldots,\ x_n : T_n \ [\leftarrow e_n] \text{ in } e$$

is defined to be the same as typing

$$\text{let } x_1 : T_1 \ [\leftarrow e_1] \text{ in } (\text{let } x_2 : T_2 \ [\leftarrow e_2], \ldots,\ x_n : T_n \ [\leftarrow e_n] \text{ in } e\ )$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O[T_1/x_1], M, C \vdash e_1 : T_1' \\ \vdots \\ O[T_n/x_n], M, C \vdash e_n : T_n' \end{array}}{O, M, C \vdash \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \ldots x_n : T_n \Rightarrow e_n; \text{ esac} : \bigsqcup_{1 \le i \le n} T_i'} \quad \text{[Case]}$$

Each branch of a `case` is type checked in an environment where variable x_i has type T_i. The type of the entire `case` is the join of the types of its branches. The variables declared on each branch of a `case` must all have distinct types.

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : \mathit{Bool} \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \mathit{Object}} \quad \text{[Loop]}$$

The predicate of a loop must have type *Bool*; the type of the entire loop is always *Object*. An `isvoid` test has type *Bool*:

$$\frac{O, M, C \vdash e_1 : T_1}{O, M, C \vdash \text{isvoid } e_1 : \mathit{Bool}} \quad \text{[Isvoid]}$$

With the exception of the rule for equality, the type checking rules for the primitive logical, comparison, and arithmetic operations are easy.

$$\frac{O, M, C \vdash e_1 : \mathit{Bool}}{O, M, C \vdash \neg e_1 : \mathit{Bool}} \quad \text{[Not]}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : \mathit{Int} \\ O, M, C \vdash e_2 : \mathit{Int} \\ \mathit{op} \in \{<, \le\} \end{array}}{O, M, C \vdash e_1 \ \mathit{op} \ e_2 : \mathit{Bool}} \quad \text{[Compare]}$$

$$\frac{O, M, C \vdash e_1 : \mathit{Int}}{O, M, C \vdash \tilde{} e_1 : \mathit{Int}} \quad \text{[Neg]}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : \mathit{Int} \\ O, M, C \vdash e_2 : \mathit{Int} \\ \mathit{op} \in \{*, +, -, /\} \end{array}}{O, M, C \vdash e_1 \ \mathit{op} \ e_2 : \mathit{Int}} \quad \text{[Arith]}$$

The wrinkle in the rule for equality is that any types may be freely compared except `Int, String` and `Bool`, which may only be compared with objects of the same type.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{\mathit{Int}, \mathit{String}, \mathit{Bool}\} \vee T_2 \in \{\mathit{Int}, \mathit{String}, \mathit{Bool}\} \Rightarrow T_1 = T_2 \end{array}}{O, M, C \vdash e_1 = e_2 : \mathit{Bool}} \quad \text{[Equal]}$$

The final cases are type checking rules for attributes and methods. For a class C, let the object environment O_C give the types of all attributes of C (including any inherited attributes). More formally, if x is an attribute (inherited or not) of C, and the declaration of x is $x : T$, then

$$O_C(x) = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T = \texttt{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}$$

The method environment M is global to the entire program and defines for every class C the signatures of all of the methods of C (including any inherited methods).

The two rules for type checking attribute defininitions are similar the rules for `let`. The essential difference is that attributes are visible within their initialization expressions. Note that `self` is bound in the initialization.

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C[\texttt{SELF\_TYPE}_C/\mathit{self}], M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C, M, C \vdash x : T_0 \leftarrow e_1;} \quad \text{[Attr-Init]}$$

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T;} \quad \text{[Attr-No-Init]}$$

The rule for typing methods checks the body of the method in an environment where O_C is extended with bindings for the formal parameters and `self`. The type of the method body must conform to the declared return type.

$$\frac{\begin{array}{l} M(C, f) = (T_1, \ldots, T_n, T_0) \\ O_C[\texttt{SELF\_TYPE}_C/\mathit{self}][T_1/x_1] \ldots [T_n/x_n], M, C \vdash e : T_0' \\ T_0' \leq \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \end{array}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \ \{ \ e \ \};} \quad \text{[Method]}$$

13 Operational Semantics

This section contains a mostly formal presentation of the operational semantics for the Cool language. The operational semantics define for every Cool expression what value it should produce in a given context. The context has three components: an environment, a store, and a self object. These components are described in the next section. Section 13.2 defines the syntax used to refer to Cool objects, and Section 13.3 defines the syntax used to refer to class definitions.

Keep in mind that a formal semantics is a specification only—it does not describe an implementation. The purpose of presenting the formal semantics is to make clear all the details of the behavior of Cool expressions. How this behavior is implemented is another matter.

13.1 Environment and the Store

Before we can present a semantics for Cool we need a number of concepts and a considerable amount of notation. An *environment* is a mapping of variable identifiers to *locations*. Intuitively, an environment tells us for a given identifier the address of the memory location where that identifier's value is stored. For a given expression, the environment must assign a location to all identifiers to which the expression may refer. For the expression, e.g., $a + b$, we need an environment that maps a to some location and b to some location. We'll use the following syntax to describe environments, which is very similar to the syntax of type assumptions used in Section 12.

$$E = [a : l_1, b : l_2]$$

This environment maps a to location l_1, and b to location l_2.

The second component of the context for the evaluation of an expression is the *store* (memory). The store maps locations to values, where values in Cool are just objects. Intuitively, a store tells us what value is stored in a given memory location. For the moment, assume all values are integers. A store is similar to an environment:

$$S = [l_1 \rightarrow 55, l_2 \rightarrow 77]$$

This store maps location l_1 to value 55 and location l_2 to value 77.

Given an environment and a store, the value of an identifier can be found by first looking up the location that the identifier maps to in the environment and then looking up the location in the store.

$$\begin{aligned} E(a) &= l_1 \\ S(l_1) &= 55 \end{aligned}$$

Together, the environment and the store define the execution state at a particular step of the evaluation of a Cool expression. The double indirection from identifiers to locations to values allows us to model variables. Consider what happens if the value 99 is assigned variable a in the environment and store defined above. Assigning to a variable means changing the value to which it refers but not its location. To perform the assignment, we look up the location for a in the environment E and then change the mapping for the obtained location to the new value, giving a new store S'.

$$\begin{aligned} E(a) &= l_1 \\ S' &= S[99/l_1] \end{aligned}$$

The syntax $S[v/l]$ denotes a new store that is identical to the store S, except that S' maps location l to value v. For all locations l' where $l' \neq l$, we still have $S'(l') = S(l')$.

The store models the contents of memory of the computer during program execution. Assigning to a variable modifies the store.

There are also situations in which the environment is modified. Consider the following Cool fragment:

```
let c : Int <- 33 in
  c
```

When evaluating this expression, we must introduce the new identifier c into the environment before evaluating the body of the `let`. If the current environment and state are E and S, then we create a new environment E' and a new store S' defined by:

$$\begin{aligned} l_c &= newloc(S) \\ E' &= E[l_c/c] \\ S' &= S[33/l_c] \end{aligned}$$

The first step is to allocate a location for the variable c. The location should be fresh, meaning that the current store does not have a mapping for it. The function $newloc()$ applied to a store gives us an unused location in that store. We then create a new environment E', which maps c to l_c but also contains all of the mappings of E for identifiers other than c. Note that if c already has a mapping in E, the new environment E' hides this old mapping. We must also update the store to map the new location to a value. In this case l_c maps to the value 33, which is the initial value for c as defined by the let-expression.

The example in this subsection oversimplifies Cool environments and stores a bit, because simple integers are not Cool values. Even integers are full-fledged objects in Cool.

13.2 Syntax for Cool Objects

Every Cool value is an object. Objects contain a list of named attributes, a bit like records in C. In addition, each object belongs to a class. We use the following syntax for values in Cool:

$$v = X(a_1 = l_1, a_2 = l_2, \ldots, a_n = l_n)$$

Read the syntax as follows: The value v is a member of class X containing the attributes $a_1, \ldots, a_n$ whose locations are $l_1, \ldots, l_n$. Note that the attributes have an associated location. Intuitively this means that there is some space in memory reserved for each attribute.

For base objects of Cool (i.e., `Int`s, `String`s, and `Bool`s) we use a special case of the above syntax. Base objects have a class name, but their attributes are not like attributes of normal classes, because they cannot be modified. Therefore, we describe base objects using the following syntax:

$$Int(5)$$
$$Bool(true)$$
$$String(4, \texttt{"}Cool\texttt{"})$$

For `Int`s and `Bool`s, the meaning is obvious. `String`s contain two parts, the length and the actual sequence of ASCII characters.

13.3 Class definitions

In the rules presented in the next section, we need a way to refer to the definitions of attributes and methods for classes. Suppose we have the following Cool class definition:

```
class B {
   s : String <- "Hello";
   g (y:String) : Int {
      y.concat(s)
   };
   f (x:Int) : Int {
      x+1
   };
};

class A inherits B {
   a : Int;
   b : B <- new B;
   f(x:Int) : Int {
      x+a
   };
};
```

Two mappings, called *class* and *implementation*, are associated with class definitions. The *class* mapping is used to get the attributes, as well as their types and initializations, of a particular class:

$$class(A) = (s : String \leftarrow \texttt{"}Hello\texttt{"},\ a : Int \leftarrow 0,\ b : B \leftarrow new\ B)$$

Note that the information for class A contains everything that it inherited from class B, as well as its own definitions. If B had inherited other attributes, those attributes would also appear in the information for A. The attributes are listed in the order they are inherited and then in source order: all the attributes from the greatest ancestor are listed first in the order in which they textually appear, then the attributes of the next greatest ancestor, and so on, on down to the attributes defined in the particular class. We rely on this order in describing how new objects are initialized.

The general form of a class mapping is:

$$class(X) = (a_1 : T_1 \leftarrow e_1,\ \ldots,\ a_n : T_n \leftarrow e_n)$$

Note that every attribute has an initializing expression, even if the Cool program does not specify one for each attribute. The default initialization for a variable or attribute is the *default* of its type. The default of `Int` is 0, the default of `String` is `""`, the default of `Bool` is `false`, and the default of any other type is `void`.[4] The default of type T is written D_T.

The implementation mapping gives information about the methods of a class. For the above example, *implementation* of A is defined as follows:

$$\begin{aligned} implementation(A, f) &= (x, x + a) \\ implementation(A, g) &= (y, y.concat(s)) \end{aligned}$$

In general, for a class X and a method m,

$$implementation(X, m) = (x_1, x_2, \ldots, x_n, e_{body})$$

specifies that method m when invoked from class X, has formal parameters $x_1, \ldots, x_n$, and the body of the method is expression e_{body}.

13.4 Operational Rules

Equipped with environments, stores, objects, and class definitions, we can now attack the operational semantics for Cool. The operational semantics is described by rules similar to the rules used in type checking. The general form of the rules is:

$$\frac{\vdots}{so, S, E \vdash e_1 : v, S'}$$

The rule should be read as: In the context where *self* is the object so, the store is S, and the environment is E, the expression e_1 evaluates to object v and the new store is S'. The dots above the horizontal bar stand for other statements about the evaluation of sub-expressions of e_1.

Besides an environment and a store, the evaluation context contains a self object so. The self object is just the object to which the identifier `self` refers if `self` appears in the expression. We do not place `self` in the environment and store because `self` is not a variable—it cannot be assigned to. Note that the rules specify a new store after the evaluation of an expression. The new store contains all changes to memory resulting as side effects of evaluating expression e_1.

[4] A tiny point: We are allowing `void` to be used as an expression here. There is no expression for `void` available to Cool programmers.

The rest of this section presents and briefly discusses each of the operational rules. A few cases are not covered; these are discussed at the end of the section.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ E(Id) = l_1 \\ S_3 = S_2[v_1/l_1] \end{array}}{so, S_1, E \vdash Id \leftarrow e_1 : v_1, S_3} \quad \text{[Assign]}$$

An assignment first evaluates the expression on the right-hand side, yielding a value v_1. This value is stored in memory at the address for the identifier.

The rules for identifier references, `self`, and constants are straightforward:

$$\frac{\begin{array}{l} E(Id) = l \\ S(l) = v \end{array}}{so, S, E \vdash Id : v, S} \quad \text{[Var]}$$

$$\frac{}{so, S, E \vdash \text{self} : so, S} \quad \text{[Self]}$$

$$\frac{}{so, S, E \vdash \text{true} : Bool(true), S} \quad \text{[True]}$$

$$\frac{}{so, S, E \vdash \text{false} : Bool(false), S} \quad \text{[False]}$$

$$\frac{i \text{ is an integer constant}}{so, S, E \vdash i : Int(i), S} \quad \text{[Int]}$$

$$\frac{\begin{array}{l} s \text{ is a string constant} \\ l = length(s) \end{array}}{so, S, E \vdash s : String(l, s), S} \quad \text{[String]}$$

A `new` expression is more complicated than one might expect:

$$\frac{\begin{array}{l} T_0 = \begin{cases} X & \text{if } T = \texttt{SELF\_TYPE} \text{ and } so = X(\ldots) \\ T & \text{otherwise} \end{cases} \\ class(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \\ l_i = newloc(S_1), \text{ for } i = 1 \ldots n \text{ and each } l_i \text{ is diistinct} \\ v_1 = T_0(a_1 = l_1, \ldots, a_n = l_n) \\ S_2 = S_1[D_{T_1}/l_1, \ldots, D_{T_n}/l_n] \\ v_1, S_2, [a_1 : l_1, \ldots, a_n : l_n] \vdash \{a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n;\} : v_2, S_3 \end{array}}{so, S_1, E \vdash \text{new } T : v_1, S_3} \quad \text{[New]}$$

The tricky thing in a `new` expression is to initialize the attributes in the right order. Note also that, during initialization, attributes are bound to the default of the appropriate class.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \\ so, S_{n+1}, E \vdash e_0 : v_0, S_{n+2} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \\ implementation(X, f) = (x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+2}), \text{ for } i = 1 \ldots n \text{ and each } l_{x_i} \text{ is distinct} \\ S_{n+3} = S_{n+2}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ v_0, S_{n+3}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+4} \end{array}}{so, S_1, E \vdash e_0.f(e_1, \ldots, e_n) : v_{n+1}, S_{n+4}} \quad \text{[Dispatch]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \\ so, S_{n+1}, E \vdash e_0 : v_0, S_{n+2} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \\ implementation(T, f) = (x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+2}), \text{ for } i = 1 \ldots n \text{ and each } l_{x_i} \text{ is distinct} \\ S_{n+3} = S_{n+2}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ v_0, S_{n+3}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+4} \end{array}}{so, S_1, E \vdash e_0@T.f(e_1, \ldots, e_n) : v_{n+1}, S_{n+4}} \quad \text{[StaticDispatch]}$$

The two dispatch rules do what one would expect. The arguments are evaluated and saved. Next, the expression on the left-hand side of the "." is evaluated. In a normal dispatch, the class of this expression is used to determine the method to invoke; otherwise the class is specified in the dispatch itself.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \end{array}}{so, S_1, E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : v_2, S_3} \quad \text{[If-True]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(false), S_2 \\ so, S_2, E \vdash e_3 : v_3, S_3 \end{array}}{so, S_1, E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : v_3, S_3} \quad \text{[If-False]}$$

There are no surprises in the if-then-else rules. Note that value of the predicate is a `Bool` object, not a boolean.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \end{array}}{so, S_1, E \vdash \{\ e_1; e_2; \ldots; e_n;\ \} : v_n, S_{n+1}} \quad \text{[Sequence]}$$

Blocks are evaluated from the first expression to the last expression, in order. The result is the result of the last expression.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ l_1 = newloc(S_2) \\ S_3 = S_2[v_1/l_1] \\ E' = E[l_1/Id] \\ so, S_3, E' \vdash e_2 : v_2, S_4 \end{array}}{so, S_1, E \vdash \text{let } Id : T_1 \leftarrow e_1 \text{ in } e_2 : v_2, S_4} \quad \text{[Let]}$$

A `let` evaluates any initialization code, assigns the result to the variable at a fresh location, and evaluates the body of the `let`. (If there is no initialization, the variable is initialized to the default value of T_1.) We give the operational semantics only for the case of `let` with a single variable. The semantics of a multiple `let`

$$\text{let } x_1 : T_1 \leftarrow e_1,\ x_2 : T_2 \leftarrow e2, \ldots,\ x_n : T_n \leftarrow e_n \text{ in } e$$

is defined to be the same as

$$\text{let } x_1 : T_1 \leftarrow e_1 \text{ in (let } x_2 : T_2 \leftarrow e_2, \ldots,\ x_n : T_n \leftarrow e_n \text{ in } e\ )$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_0 : v_0, S_2 \\ v_0 = X(\ldots) \\ T_i = \text{closest ancestor of } X \text{ in } \{T_1, \ldots, T_n\} \\ l_0 = newloc(S_2) \\ S_3 = S_2[v_0/l_0] \\ E' = E[l_0/Id_i] \\ so, S_3, E' \vdash e_i : v_1, S_4 \end{array}}{so, S_1, E \vdash \text{case } e_0 \text{ of } Id_1 : T_1 \Rightarrow e_1; \ldots; Id_n : T_n \Rightarrow e_n;\ \text{esac} : v_1, S_4} \quad \text{[Case]}$$

Note that the `case` rule requires that the class hierarchy be available in some form at runtime, so that the correct branch of the `case` can be selected. This rule is otherwise straightforward.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_4} \quad \text{[Loop-True]}$$

$$\frac{so, S_1, E \vdash e_1 : Bool(false), S_2}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_2} \quad \text{[Loop-False]}$$

There are two rules for `while`: one for the case where the predicate is `true` and one for the case where the predicate is `false`. Both cases are straightforward. The two rules for `isvoid` are also straightforward:

$$\frac{so, S_1, E \vdash e_1 : void, S_2}{so, S_1, E \vdash \text{isvoid } e_1 : Bool(true), S_2} \quad \text{[IsVoid-True]}$$

$$\frac{so, S_1, E \vdash e_1 : X(\ldots), S_2}{so, S_1, E \vdash \text{isvoid } e_1 : Bool(false), S_2} \quad \text{[IsVoid-False]}$$

The remainder of the rules are for the primitive arithmetic, logical, and comparison operations except equality. These are all easy rules.

$$\frac{so, S_1, E \vdash e_1 : Bool(b), S_2 \quad v_1 = Bool(\neg b)}{so, S_1, E \vdash \text{not } e_1 : v_1, S_2} \quad \text{[Not]}$$

$$\frac{so, S_1, E \vdash e_1 : Int(i_1), S_2 \quad so, S_2, E \vdash e_2 : Int(i_2), S_3 \quad op \in \{\leq, <\} \quad v_1 = \begin{cases} Bool(true), & \text{if } i_1 \ op \ i_2 \\ Bool(false), & \text{otherwise} \end{cases}}{so, S_1, E \vdash e_1 \ op \ e_2 : v_1, S_3} \quad \text{[Comp]}$$

$$\frac{so, S_1, E \vdash e_1 : Int(i_1), S_2 \quad v_1 = Int(-i_1)}{so, S_1, E \vdash \tilde{} e_1 : v_1, S_2} \quad \text{[Neg]}$$

$$\frac{so, S_1, E \vdash e_1 : Int(i_1), S_2 \quad so, S_2, E \vdash e_2 : Int(i_2), S_3 \quad op \in \{*, +, -, /\} \quad v_1 = Int(i_1 \ op \ i_2)}{so, S_1, E \vdash e_1 \ op \ e_2 : v_1, S_3} \quad \text{[Arith]}$$

Cool `Int`s are 32-bit two's complement signed integers; the arithmetic operations are defined accordingly.

The notation and rules given above are not powerful enough to describe how objects are tested for equality, or how runtime exceptions are handled. For these cases we resort to an English description.

In $e_1 = e_2$, first e_1 is evaluated and then e_2 is evaluated. The two objects are compared for equality by first comparing their pointers (addresses). If they are the same, the objects are equal. The value `void` is not equal to any object except itself. If the two objects are of type `String`, `Bool`, or `Int`, their respective contents are compared.

In addition, the operational rules do not specify what happens in the event of a runtime error. Execution aborts when a runtime error occurs. The following list specifies all possible runtime errors.

1. A dispatch (static or dynamic) on `void`.
2. A case on `void`.
3. Execution of a case statement without a matching branch.
4. Division by zero.
5. Substring out of range.
6. Heap overflow.

Finally, the rules given above do not explain the execution behaviour for dispatches to primitive methods defined in the `Object`, `IO`, or `String` classes. Descriptions of these primitive methods are given in Sections 8.3-8.5.

14 Acknowledgements

Cool is based on Sather164, which is itself based on the language Sather. Portions of this document were cribbed from the Sather164 manual; in turn, portions of the Sather164 manual are based on Sather documentation written by Stephen M. Omohundro.

A number people have contributed to the design and implementation of Cool, including Manuel Fähndrich, David Gay, Douglas Hauge, Megan Jacoby, Tendo Kayiira, Carleton Miyamoto, and Michael Stoddart. Joe Darcy updated Cool to the current version.

WHAT IS CLAIMED IS

1. A method of generating an instruction in a first object-oriented computer language for operating in a decentralized computer network from an instruction in a second object-oriented computer language that can operate on a local computer, where a semantic of the second object-oriented computer language can be expressed by a rule

so, S, E, ⊢ e: v, S'

with ⊢ denoting a semantic relation said rule meaning that in a context where a variable self refers to an object *so*, where a data store on the local computer is S, and where an execution environment is E, an expression *e* evaluates to an object *v* at a second data store *S'* in the local computer, said method comprising:

replacing a local

storage update command S[v/l], which updates an at least one location l by an object v, by a command to

(a) broadcast to the decentralized computer network an at least one transaction that spends at least one unspent-transaction-output corresponding to said location l into a new output that stores at least one of (i) said object v and (ii) information enabling said object v to be determined; and

(b) spending at least one additional unspent-transaction-output corresponding to a second location l' in said execution environment E or said object so;

wherein said generated instruction enables operation of a smart contract.

2. The method of claim 1, wherein said instruction in said first object-oriented computer language is a storage-update instruction, and wherein said storage-update instruction either immediately precedes a recursive call to a first semantic relation or is located at an end of a body of said semantic rule of said first object-oriented computer language.

3. A method of claim 1, wherein said instruction in said first object-oriented computer language is an instruction for storing a data structure *D* at an unspent memory location *OUT* on the decentralized computer network, said data structure *D* further including a data structure *D'*, where said data structure D' further includes a data structure *D''*, said method comprising the steps of:

(a) receiving at the local computer a request to store the data structure *D* at an unspent memory location *OUT* on the decentralized computer network;

(b) creating at the local computer a request *TX*, to the decentralized computer network, to spend the unspent memory location *OUT* and to generate a new unspent memory location *OUT'* on the decentralized computer network;

(c) using the local computer to sign and broadcast said request *TX* to the decentralized computer network;

(d) receiving said request *TX* at the decentralized computer network;

(e) using the decentralized computer network to determine validity of said request *TX* according to a protocol of the decentralized computer network; and

(f) if said request *TX* is determined to be valid, performing the steps of

(i) spending the unspent memory location *OUT* on the decentralized computer network and generating said new unspent memory locations *OUT'* on the decentralized computer network; and

(ii) repeating steps (a) through (f)(i) above while using said data structure *D'* in place of said data structure *D* and while using said memory location *OUT'* in place of said memory location *OUT*.

4. The method of claim 3, wherein the decentralized computer network is an unspent-transaction-outputs blockchain.
5. The method of claim 1, wherein the decentralized computer network is an unspent-transaction-outputs blockchain.

6. The method of claim 1, wherein a source code of a program written in one of said first computer language and said second computer language is stored in a transaction. that

7. The method of claim 6, where said transaction represents object creation.

8. A computer system comprising a local computer, the local computer comprising:

a) a memory for storing computer instruction and data; and

b) a processor operatively coupled to said memory, said processor capable of generating an instruction in a first object-oriented computer language for operating in a decentralized computer network from an instruction in a second object-oriented computer language that can operate on the local computer, where a semantic of the second object-oriented computer language can be expressed by a rule

so, S, E, ⊢ e: v, S'

with ⊢ denoting a semantic relation said rule meaning that in a context where a variable self refers to an object so, where a data store on the local computer is S, and where an execution environment is E, an expression *e* evaluates to an object *v* at a second data store *S'* in the local computer, said method comprising:

replacing a centralized storage update command S[v/l], which updates an at least one location l by an object v, by a command to

(c) broadcast to the decentralized computer network an at least one transaction that spends at least one unspent-transaction-output corresponding to said location l into a new output that stores at least one of (i) said object v and (ii) information enabling said object v to be determined; and

(d) spending at least one additional unspent-transaction-output corresponding to a second location l' in said execution environment E or said object so;

wherein said generated instruction enables operation of a smart contract.

9. The computer system of claim 8, wherein the decentralized computer network is an unspent-transaction-outputs blockchain.

10. The computer system of claim 8, further comprising a decentralized computer network,

wherein said instruction in said first object-oriented computer language is an instruction for storing a data structure *D* at an unspent memory location *OUT* on the decentralized computer network, said data structure *D* further including a data structure *D'*, where said data structure D' further includes a data structure *D''*,

(a) wherein the local computer is configured to receive a request to store the data structure *D* at an unspent memory location *OUT* on the decentralized computer network;

(b) wherein the local computer is configured to create a request *TX* to the decentralized computer network, to spend the unspent memory location *OUT* and to generate a new unspent memory location *OUT'* on the decentralized computer network;

(c) wherein the local computer to configured to enable signing and broadcasting said request *TX* to the decentralized computer network;

(d) wherein receiving the decentralized computer network is configured to receive said request *TX*;

(e) wherein the decentralized computer network is configured enable determining validity of said request *TX* according to a protocol of the decentralized computer network; and

(f) if said request *TX* is determined to be valid,

(i) said decentralized computer network further configured to spend the unspent memory location *OUT* on the decentralized computer network and to generate said new unspent memory locations *OUT'* on the decentralized computer network; and

(ii) wherein the computer system is configured to repeating operations (a) through (f)(i) above while using said data structure *D'* in place of

said data structure *D* and while using said memory location *OUT'* in place of said memory location *OUT*.

11. The computer system of claim 10, wherein the decentralized computer network is an unspent-transaction-outputs blockchain.

12. The computer system of claim 8, wherein a source code of a program written in one of said first computer language and said second computer language is stored in a transaction that represents object creation.

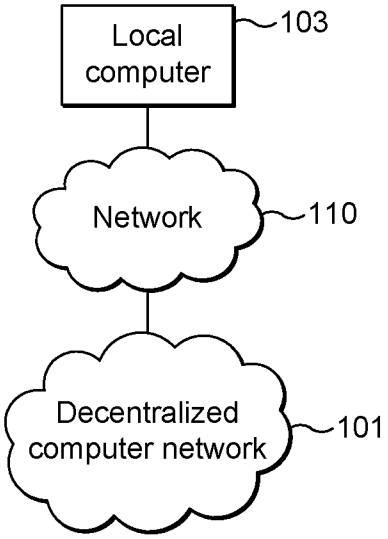
Local computer
103
Network
110
Decentralized computer network
101

FIG. 1

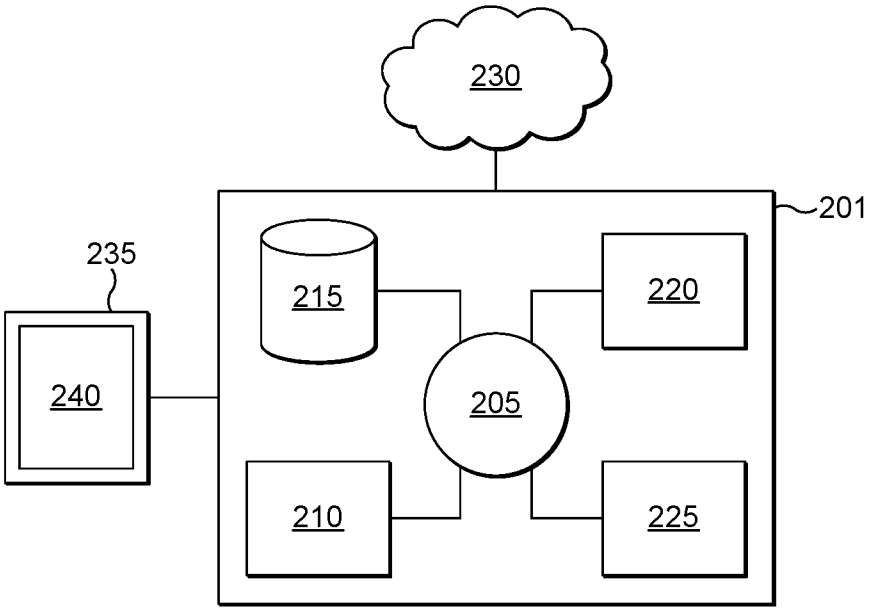
230
201
235
215
220
240
205
210
225

FIG. 2

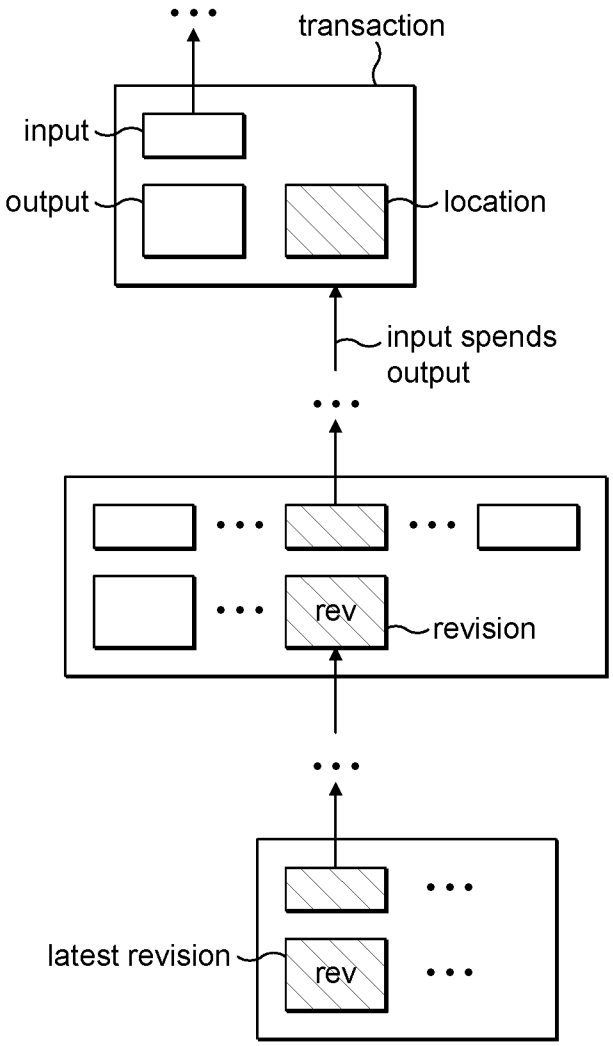
transaction
input
output
location
input spends
output
rev
revision
latest revision
rev

FIG. 3

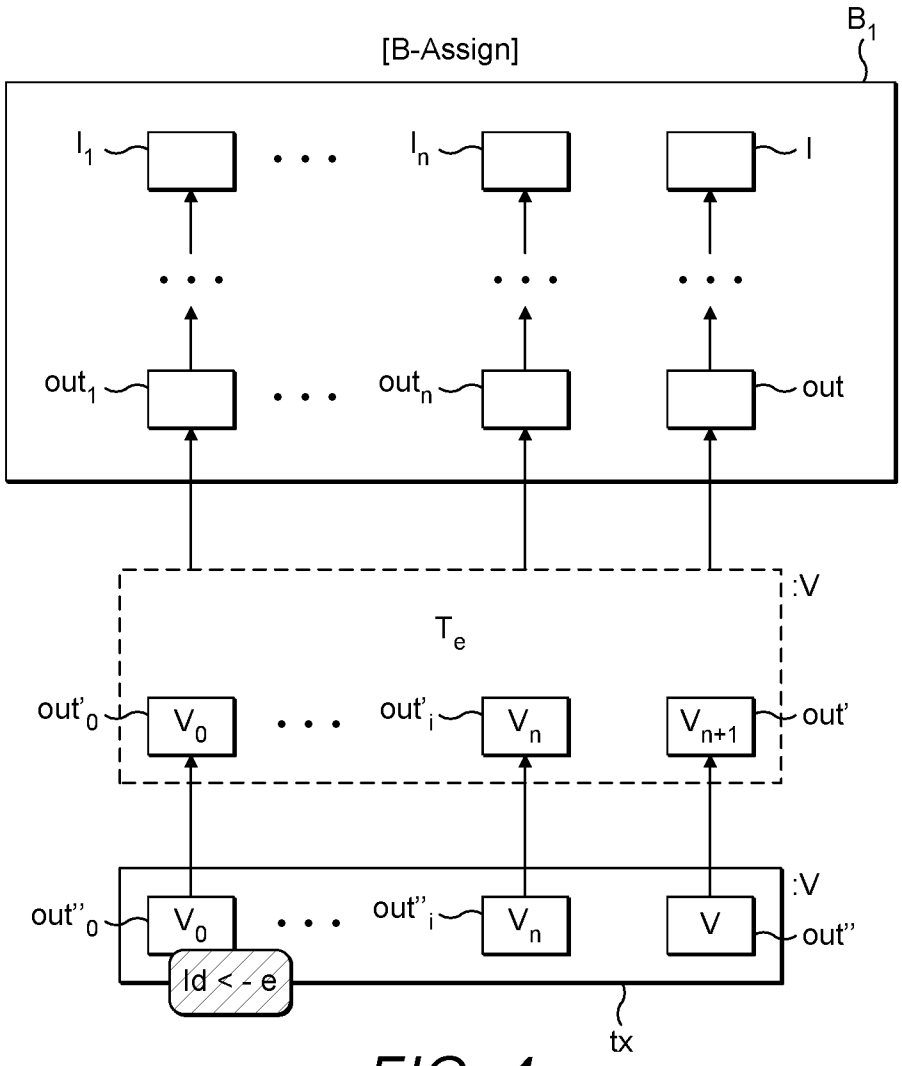
B1
[B-Assign]
I1
In
I
out1
outn
out
:V
Te
out'0
V0
out'i
Vn
Vn+1
out'
:V
out''0
V0
out''i
Vn
V
out''
Id < - e
tx

FIG. 4

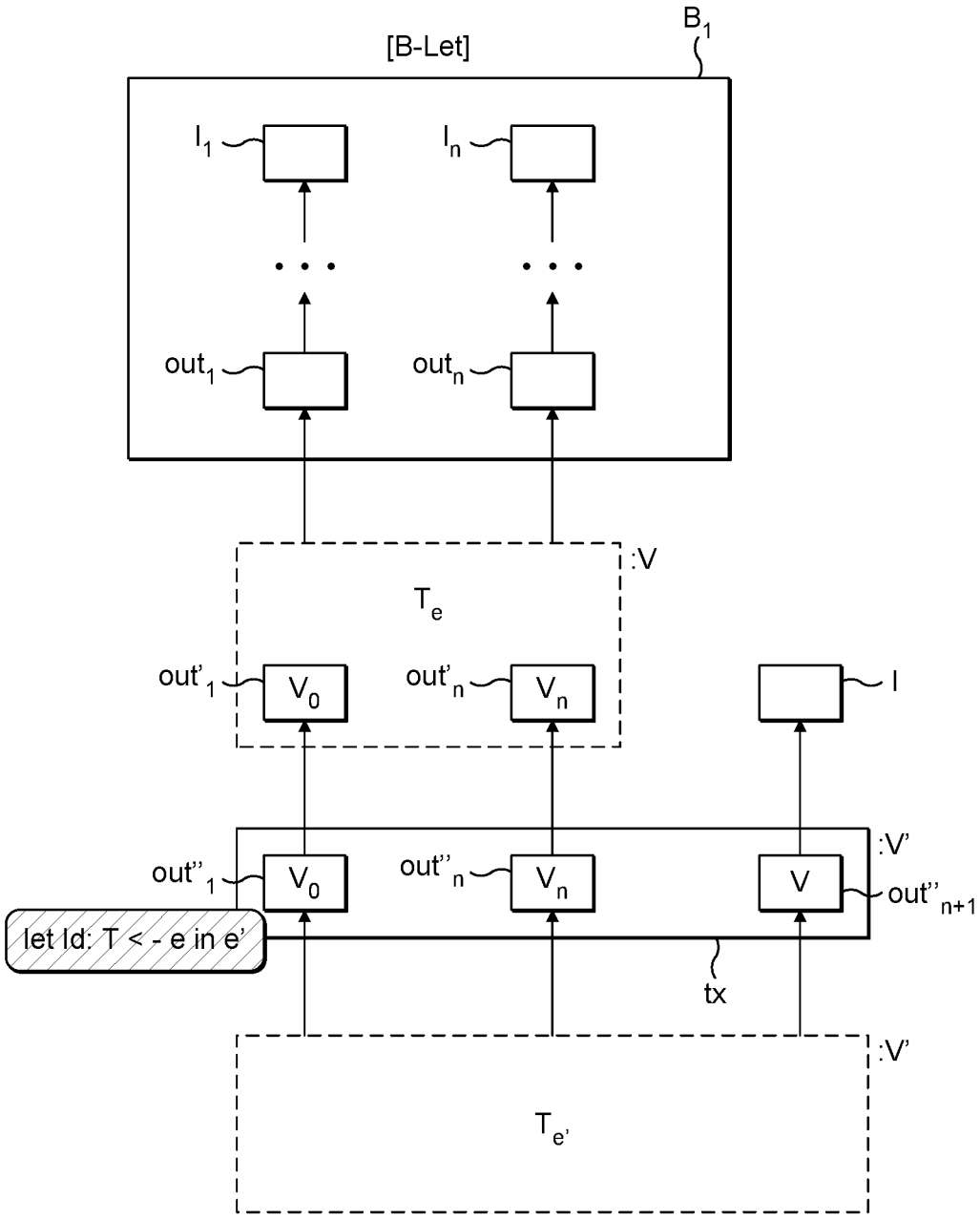
B1
[B-Let]
I1
In
out1
outn
:V
Te
out'1
V0
out'n
Vn
I
:V'
out''1
V0
out''n
Vn
V
out''n+1
let Id: T < - e in e'
tx
:V'
Te'

FIG. 5

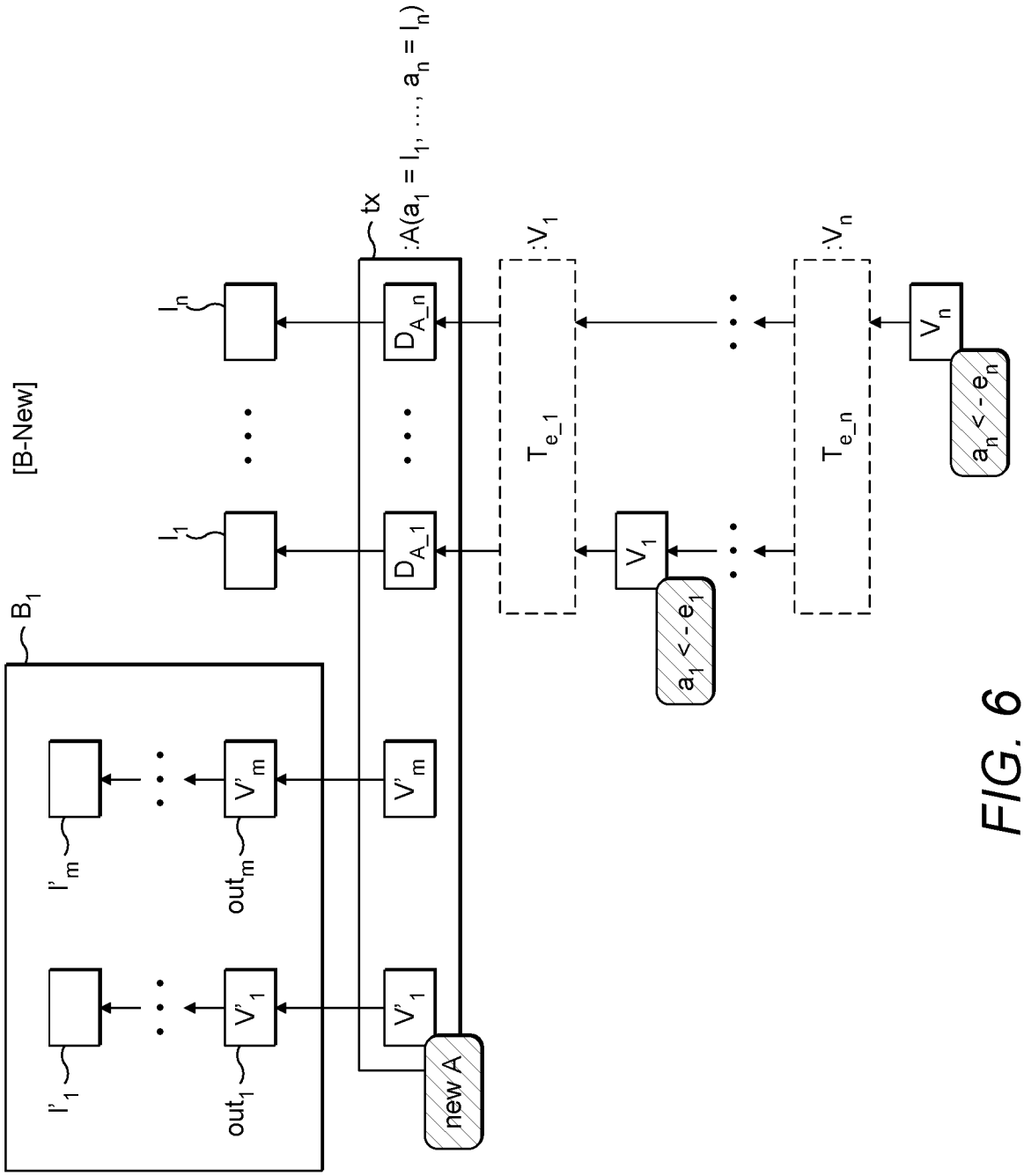
B1
[B-New]
I'1
I'm
out1
outm
V'1
V'm
new A
tx
:A(a1 = I1, ..., an = In)
I1
In
DA_1
DA_n
:V1
:Vn
Te_1
Te_n
V1
Vn
a1 < - e1
an < - en

FIG. 6

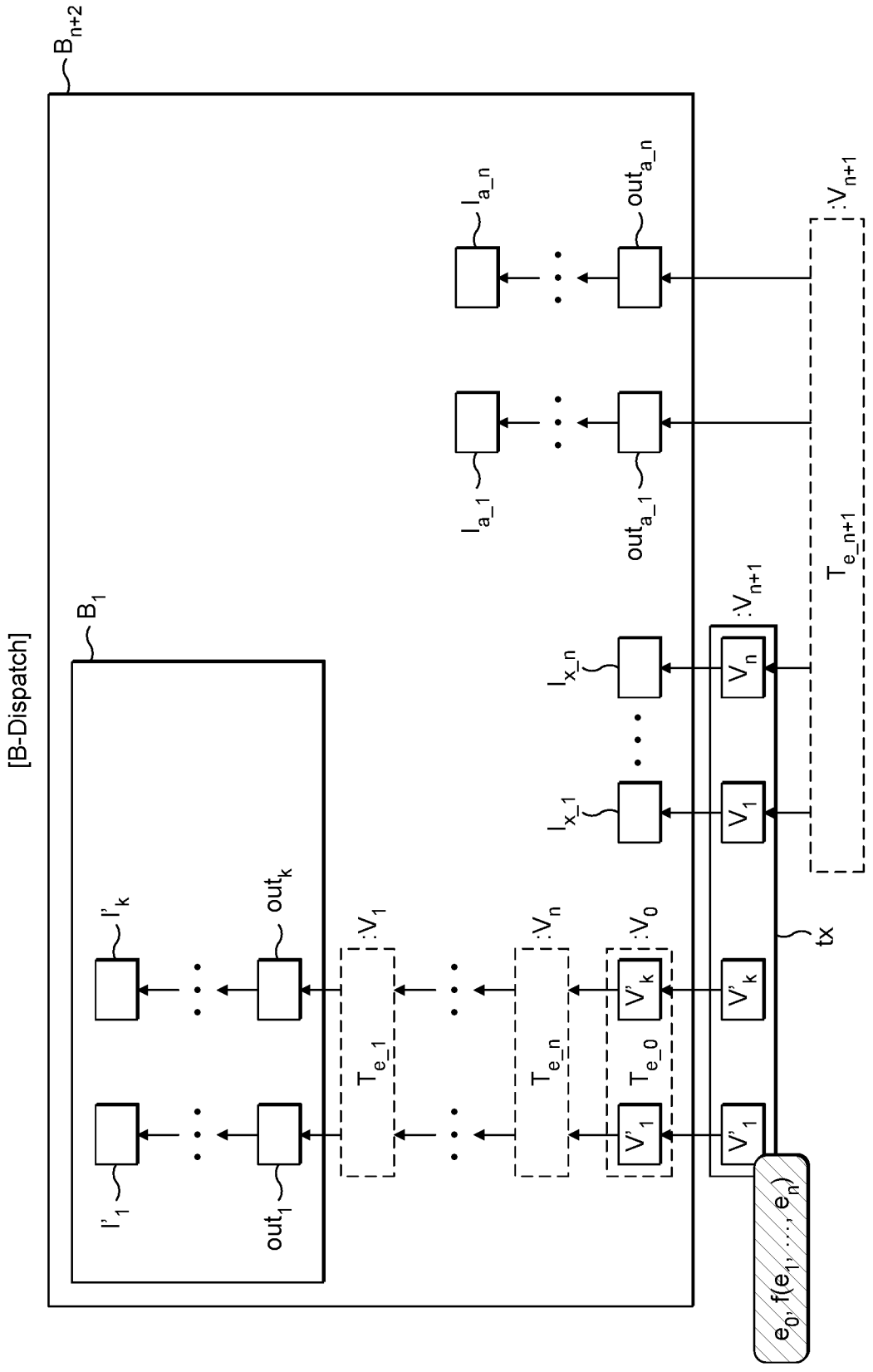
[B-Dispatch]
B_{n+2}
B_1
I'_1
I'_k
out_1
out_k
T_{e_1}
T_{e_n}
T_{e_0}
$:V_1$
$:V_n$
$:V_0$
V'_1
V'_k
I_{x_1}
I_{x_n}
I_{a_1}
I_{a_n}
out_{a_1}
out_{a_n}
V_1
V_n
$:V_{n+1}$
T_{e_n+1}
tx
$e_0, f(e_1, \ldots, e_n)$

FIG. 7

[Variable Definition Example]

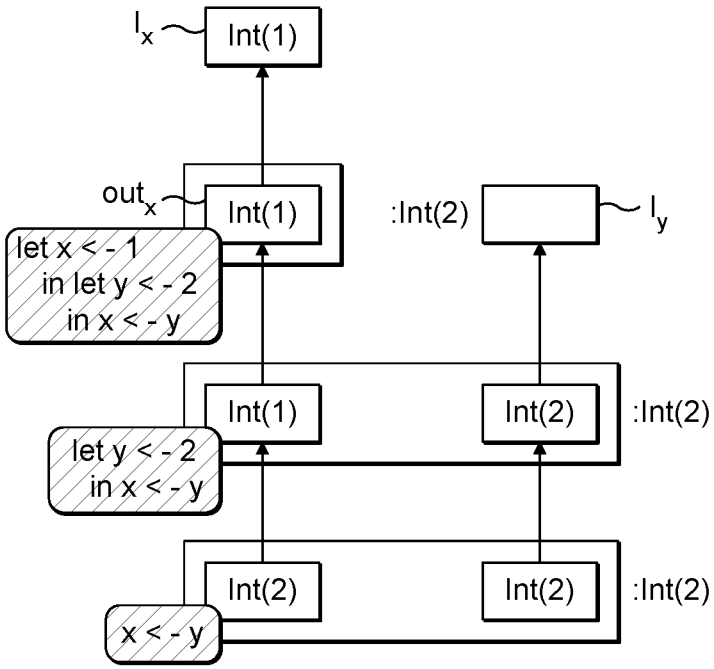

FIG. 8

[Object Creation Example 1]

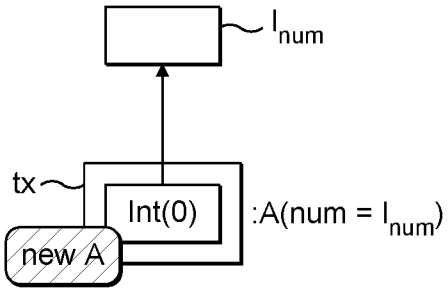

FIG. 9

[Object Creation Example 2]

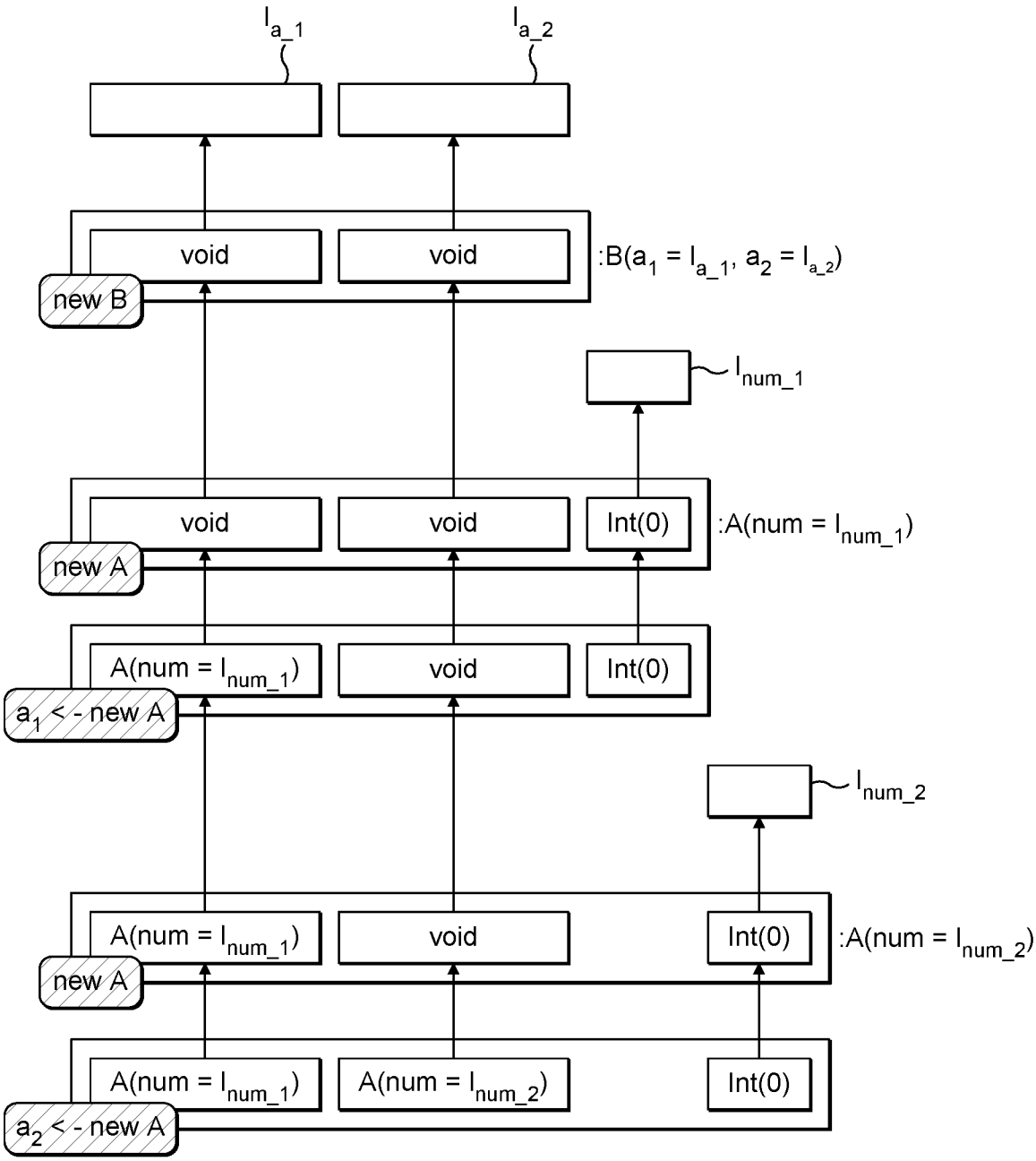

FIG. 10

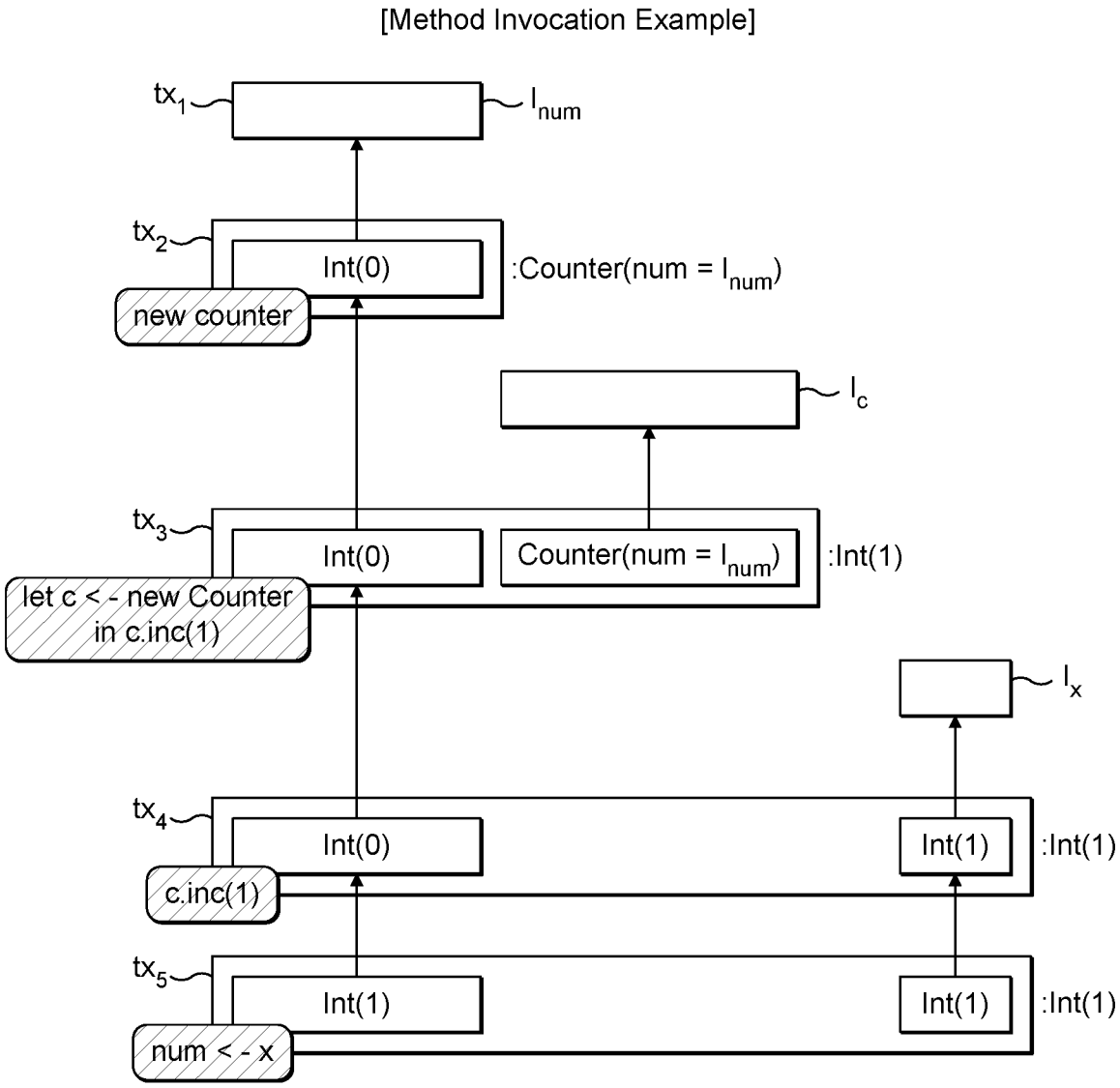
[Method Invocation Example]
tx1
lnum
tx2
Int(0)
:Counter(num = lnum)
new counter
lc
tx3
Int(0)
Counter(num = lnum)
:Int(1)
let c < - new Counter
in c.inc(1)
lx
tx4
Int(0)
Int(1)
:Int(1)
c.inc(1)
tx5
Int(1)
Int(1)
:Int(1)
num < - x

FIG. 11

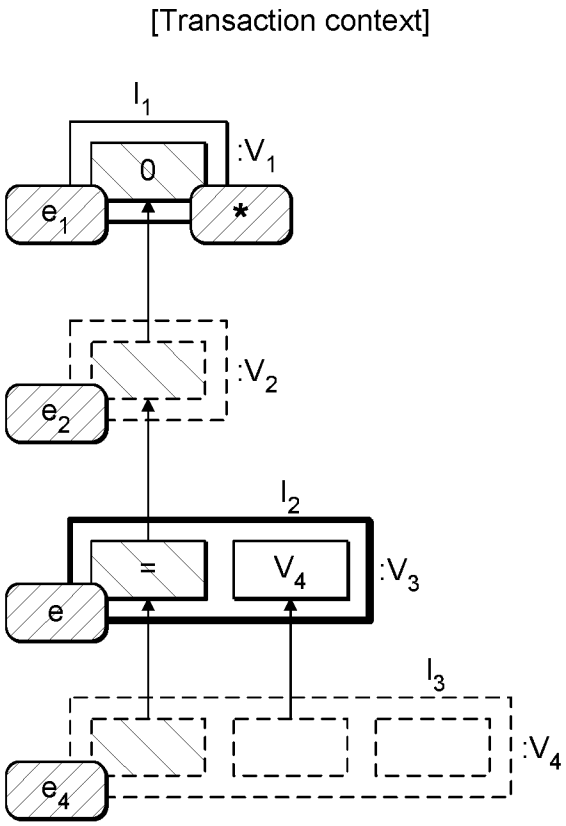

FIG. 12

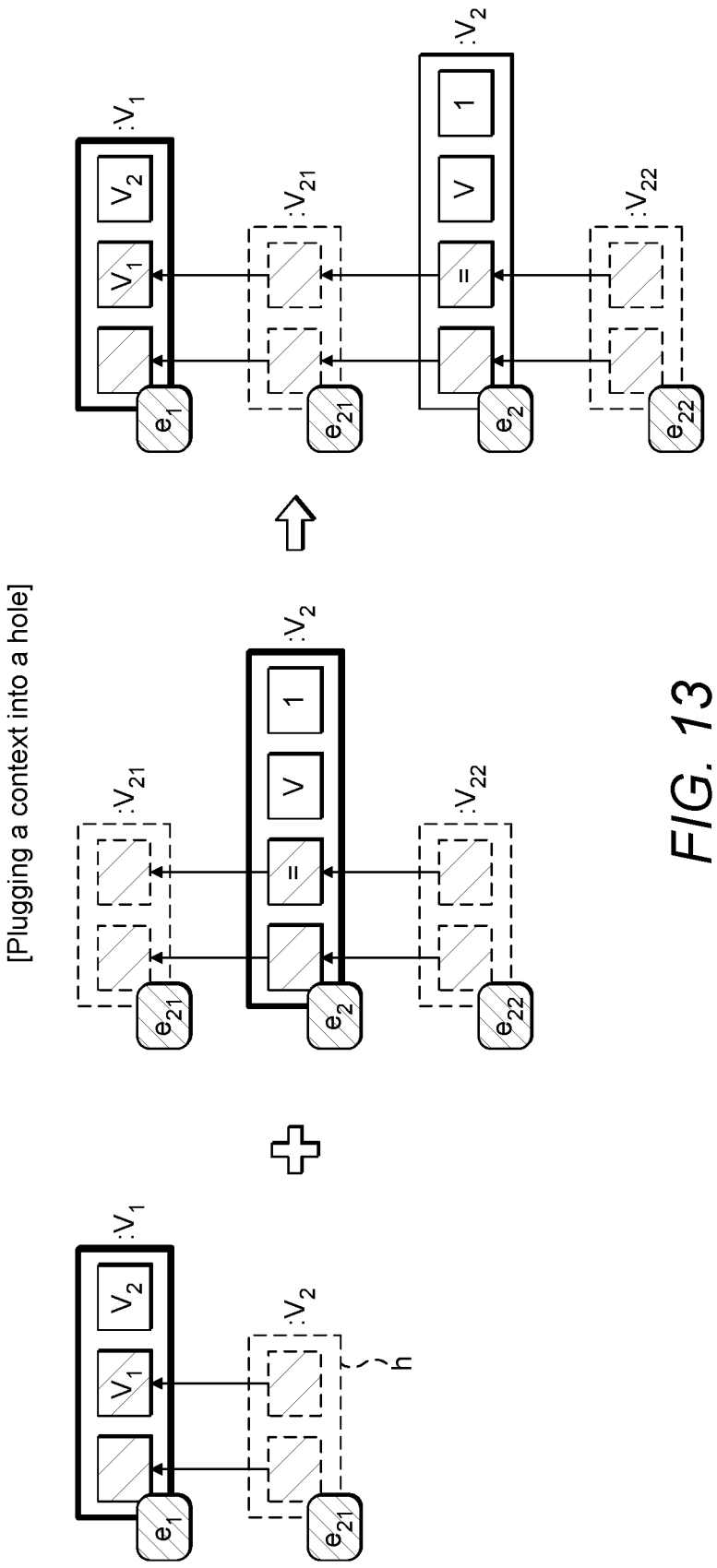
[Plugging a context into a hole]
:V1
V2
V1
e1
:V21
h
e21
+
:V2
1
V
=
e2
:V21
:V22
e21
e22
=
:V1
V2
V1
e1
:V2
1
V
e2
:V21
:V22
e21
e22

FIG. 13

[Closing a hole]

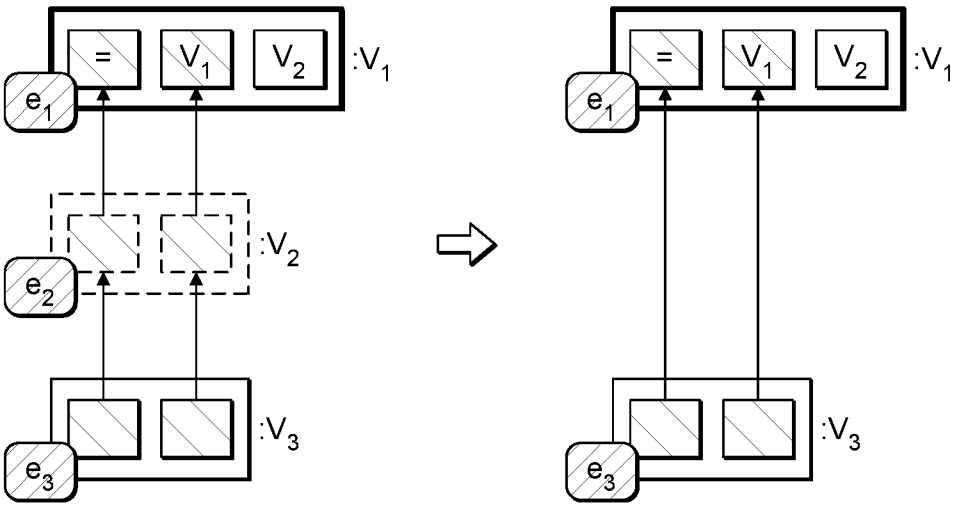

FIG. 14

[Bcool Contexts]

[Ctx-Assign]

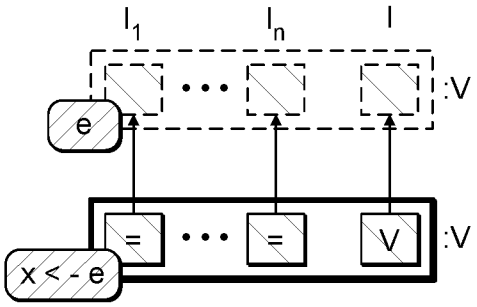

FIG. 15

[Ctx-New]

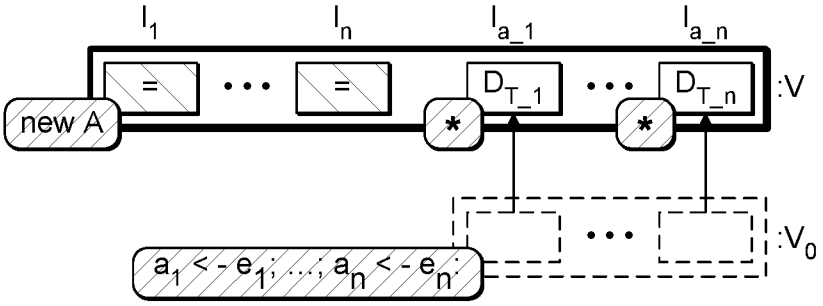

FIG. 16

[Ctx-Let]

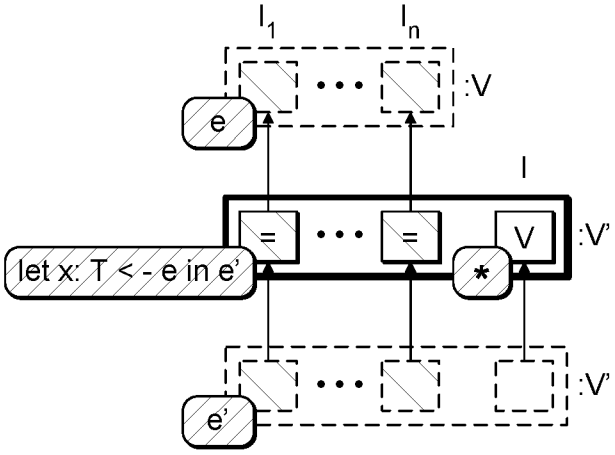

FIG. 17

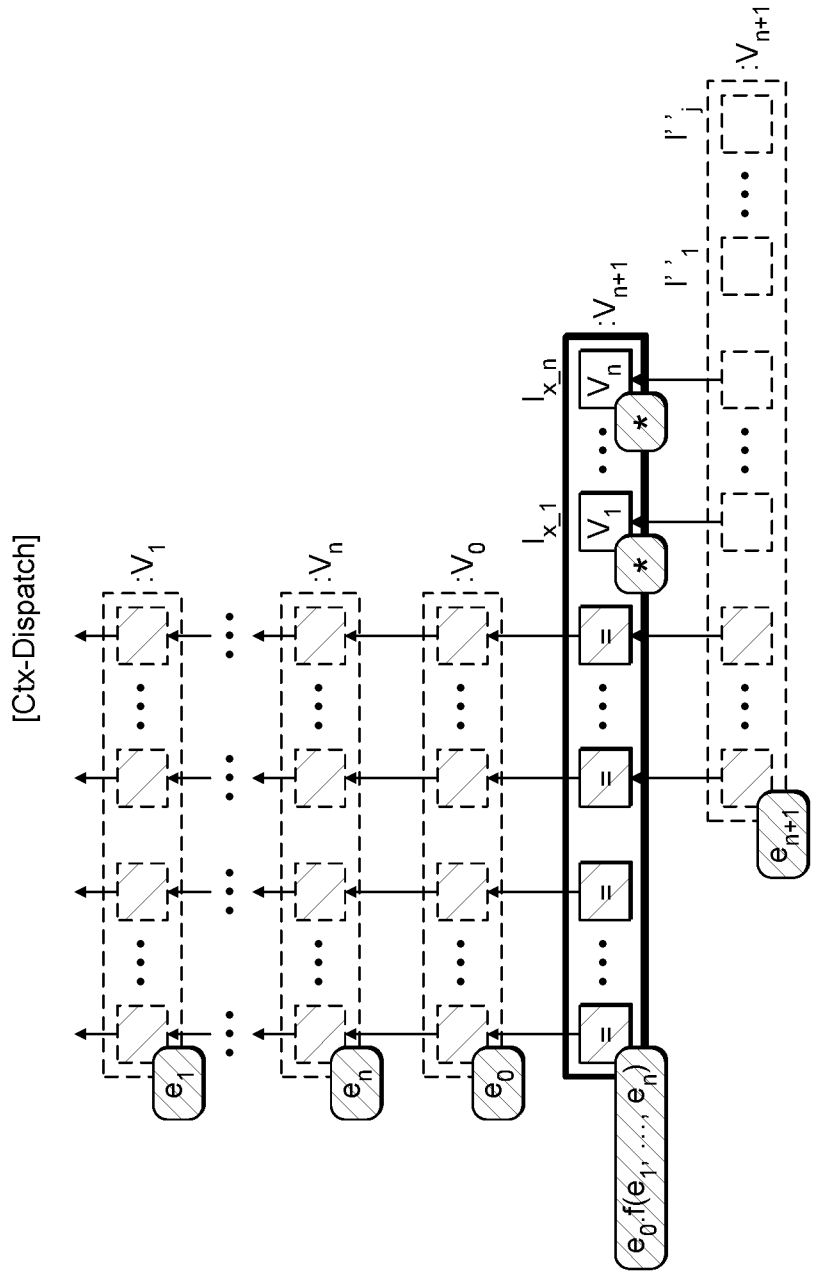
[Ctx-Dispatch]
:V1
:Vn
:V0
Ix_n
Ix_1
:Vn+1
I' '1
I' 'j
:Vn+1
Vn
V1
=
*
e1
en
e0
e0.f(e1, ..., en)
en+1

FIG. 18