The Big-Step Operational Semantics of the Code-Time Computational Model

BY

Sean Halle
seanhalle@yahoo.com

Abstract

The CodeTime Parallel Software Platform aims to allow write once run-anywhere software development for parallel computers. This goal depends upon having a format to distribute programs in which has the six properties: the code 1) is invariant to the number of animators (processors), 2) is invariant to changing the size of data, 3) allows simply and automatically changing the code-length of a schedulable work-unit, 4) allows source-languages which provide easy identification and specification of parallel tasks, 5) allows source-languages which provide simple coordination of parallel tasks, and 6) allows efficient translation to a wide variety of machine-code formats and architectures.

This paper presents the operational semantics of a format having all six of these properties. It takes the form of a circuit, with multiple isolated memory spaces which conceptually move on wires between the computational-units of the circuit.

The semantics are presented in a slightly modified form of Big Step semantics.

1 Introduction

Parallel programming is necessary to use any of the top 500 supercomputers in the world. However, this type of programming is difficult. It is especially difficult to write both correct and high performance programs.

On the other hand, all major microprocessor manufacturers are building either mult-threaded or multi-core chips for the next generation. The days of single-threaded processors are over. Over the years, the number of threads on a chip will increase exponentially.

This requires commercial software developers to write parallel programs.

A required feature for success of commercial software is hardware independence. This was proven by both the System/360 line from IBM, and the x86 instruction set from Intel. This assertion is also illustrated by the cost seen by scientific code when it is updated to run on a new super-computer.

However, no viable solution has yet been found for wide-scale, high performance, hardware independence.

The CodeTime platform solves this problem by providing a virtual-server that takes programs expressed in a format having six properties. These properties enable hardware manufacturers to write a compiler tuned to their machine. The compiler takes programs in this format and produces machine-specific binaries.

The six properties are:

1. Invariance to the number of animators. An animator is a virtual processor. This property allows the number of processors to be changed independently of the details of the code.
2. Invariance to changing the size of data. This allows tuning the granularity of task-data-size to the details of the machine, such as network speed, processor speed, and number of processors.

3. Simple and automatic change of the code-length of a schedulable work-unit. A schedulable work-unit is a set of data plus a fragment of code. Shorter run-times for work-units allow more scheduling decisions per time. Many-processor machines with fast communication benefit from short running tasks and frequent scheduling decisions that reduce the delay in the, dynamic, schedule-control-loop. Conversely, machines with few processors and slow communication will be more efficient with longer-running tasks that require less-frequent scheduling decisions.
4. Source-languages are possible which source-compile to the distribution-format and provide easy identification and specification of parallel tasks.
5. Source-languages are possible which provide simple determination and declaration of coordination of parallel tasks
6. Efficient binaries result from translation, to a wide variety of machine-code formats and architectures. This suggests semantic choices such as call-by-value rather than call-by-name, and exposing the details of memory behavior. These choices both contrast to those made in, for example, Lazy Functional Languages such as Haskell, but agree with the choices in languages such as C and Java.

In addition, the virtual-server must provide persistent storage for the result of the compilation. It therefore must also provide a user-interface to manage the persistent storage, install programs (cause them to be compiled), and run programs.

In other words, the virtual-server presents a virtual OS interface to users. This OS interface remains the same across all physical platforms. The user perceives a single machine with a fixed set of commands, irregardless of the hardware and underlying OS. All numbers of physical machines, with all manner of storage architectures are placed underneath this virtual OS. A translation layer allows the virtual server to use the machine's underlying native OS to carry out its commands.

Thus, the infrastructure to achieve hardware independence with high performance is provided by the use of an intermediate format plus hardware-specific compilers plus an OS interface. The hardware-specific compilers and OS interface portions of the CodeTime platform are detailed elsewhere.

In this paper, the formal semantics of the intermediate format are given. The format uses a circuit paradigm to represent programs. The Big Step operational semantics are extended in order to represent the behavior of circuit-elements. The semantics of the intermediate circuit format are then given in terms of this modified Big Step machinery.

2 Related Work

Large Grain Data Flow has a similar structure as CodeTime. However, CodeTime generalizes the semantics of data flow and introduces new concepts. The extensions enable several of the six properties, which classical Data-Flow and Large Grain Data Flow do not posses.

Likewise, the circuit format defined here, for CodeTime, does not have a natural physical circuit embodiment. Rather, it remains as neutral to physical processor as possible, employing such conventions as an address space containing other address spaces, circuit elements which spawn copies of themselves with an arbitrary number existing at any moment, and bags of data with undefined ordering of arrival and extraction. (As a reassurance to those concerned about implementation, simple and efficient translations of each of these features do exist, making implementation, in practice, actually easier through increased freedom of choice.)

The Java Virtual Machine defines a byte-code format, gcc uses an internal intermediate format, and abstract machines, such as for lambda calculus and PRAM, exist. However, the intermediate formats use semantics which expose the number of animators in the code. The JVM, for example, implies a single thread, with instructions that explicitly create additional threads. Each thread has exactly one animator, semantically. The lambda calculus admits some degree of variation in the number of animators if call-by-name semantics are used, but it is not amenable (without modification) to easily defining and coordinating a large number of independent parallel tasks. The shortcomings of PRAM have been detailed elsewhere.

Other parallel frameworks exist, such as MPI, CSP, and pi-calculus. These each define a means for controlling interactions between animators. For example, MPI exposes explicitly the creation of animators between which messages are sent. CSP and pi-calculus likewise define sequential processes which communicate (each process is based upon a lambda-calculus abstract machine with well-defined ordering between communication events). However, a sequential process has, semantically, a single animator. Thus, none of these have the property of code-invariance to the number of animators. By invariance is meant that the number of animators must be choosable independently of the details of the code.

On source languages, it would be nice to have a familiar language. Modified versions of most of the popular language paradigms should be possible. The extensions needed for object-oriented languages like Java and C++ have been identified. Likewise, the mechanisms needed for higher-order functions with a polymorphic type system such as in OCAML have been identified. These planned languages will have to expose the circuit nature of the distribution format, but the essential flavor of the language should stay intact, to a fair degre

3 The Elements of a CodeTime Circuit

A CodeTime circuit is composed of units connected by wires. Two kinds of unit exist, system-units and function-units.

A system-unit is a black-box to a program. It is implemented directly on specific hardware as part of implementing the Virtual Server for that hardware.

On the other hand, a function-unit is defined by the programmer. It contains three primitive elements: a coordination-element, a function-generator-element, and an output-element.

The primitive elements inside the function units perform all computation and coordination of data-movement. The system units perform all the system functions such as input and output, starting execution and ending execution.

3.1 Details of Units

Two kinds of unit exist: function-units and system-units.

3.1.1 Function Unit Details.

The first kind of unit, function-unit, performs the computation of the circuit. A function unit contains three primitive elements which are wired in series.

When data enters a function-unit, it comes off a wire and enters a pool of input-data. Each pool of input-data resides in the coordination-element. If the function-unit has several inputs, then the coordination-element will hold one input-pool for each input. The coordination-element consumes the input data to generate sets of input-parameters.

Each of these sets causes the function-generator-element to create a function-animator. The function-animator consumes the set of input parameters it was created for. The process of a function-animator consuming an input-set is the process of the computation of the program taking place. Each function-animator is a schedulable work-unit.

Upon completing animation, a function-animator produces a bag of output values. This bag is placed into a pool of output-bags. The pool of output-bags resides inside the output-element. The output-element consumes these bags and distributes the result-values in them to the input pools of other units.

3.1.2 System Unit Details.

The second kind of unit, the system-unit, is the equivalent of a system call. Several types of these units are specified. Each performs a particular system function such as input, output, name lookup,,, persistent storage, authentication, etc. They are opaque; the insides cannot be seen and are not specified. The entire unit is implemented as a single entity as part of each virtual server's implementation (system unit implementation is hardware specific). Together, these units comprise all operating system calls available in the platform.

Stucturally, only two things are defined in a system-unit. Each input to a system-unit has an input-pool attached. Each output from a system-unit places values into the attched input-pool.

3.2 Memory model

The memory is split into multiple independent address spaces, called containers. Each container is equivalent to a separate virtual-memory space.

Any container which "travels" on a wire must have a special field called a tag. A tag holds programmer-specified information. The tag information is used as the terms in the coordination-element's constraints.

A container which has a tag, and so is eligible to travel on a wire, is called a coordination container.

Containers have one or more fields. Each field may hold either a value or the name of another container.

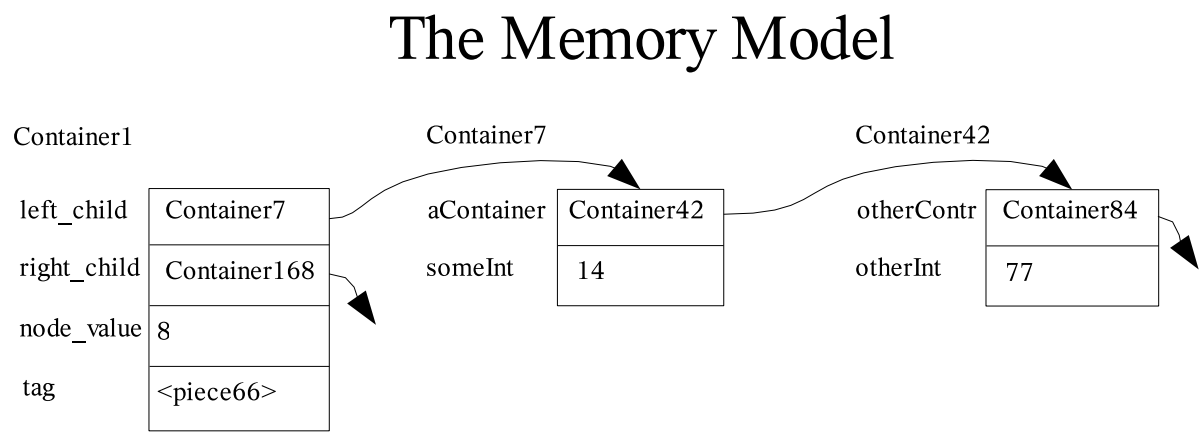

Figure 1. Each box represents a container, which is a separate address space. The labels above the boxes are the names of the address spaces. The labels to the left of each box are the addresses. To the right of each address, inside the box is the memory contents. For example, in the address space named "Container1", the address "left_child" contains the name of another container, "Container7". Container1 has a tag. The tag value is the symbol "piece66". Because Container1 has a tag, it is a special form of container, called a coordination-container.

3.3 Details of Primitive Elements Inside Function Units

Three kinds of element exist inside a function-unit. These are coordination elements, function elements, and output elements.

3.3.1 Coordination Element Details

A coordination element implements the language-processor's scheduler. See the "Command-Plus-Scheduler Model of Computation" (published by me, also as a technical report). The programmer specifies the program-level-scheduler constraints on which input-data may be grouped together. These constraints are inside the coordination element.

The coordination-element's effect is to control the order in which computations take place. In general, the order of computations (in part) is determined by which input-data is grouped with which other input-data. This concept is explained at the source-language level in "Matrix Multiply Example in BCTL"(published by me, also as a technical report).

Mechanically, the coordination-element chooses sets of containers from its various input-pools. It chooses each set such that the coordination-element's compound-constraint is satisfied.

A constraint takes the form of a boolean of terms. Each term specifies an input-pool to pick a container from. Each term also specifies some expression involving the fields of the chosen container. The expression must reduce to a boolean value. When the boolean value of each term is substituted into the constraint, a resulting value of true means the chosen containers form a valid input-set.

These constraints take the place of the various synchronisation mechanisms used in other languages. The constraints directly encode the program-level scheduler's constraints. In other languages, synchronisation mechanisms are used to construct custom program-level schedulers. Thus, the coordination-element's constraint serves the same purpose as synchronisation mechanisms.

Once an input-set is chosen, the coordination element creates an empty Function Execution Instance, then creates an empty local-store in that FEI and places the input-set into that local-store. The coordination element places this partially-filled FEI into a pool of such instances in the function-generator element.

Input sets are chosen such that no coordination-container in any input-pool will appear in more than one input-set. Also, all coordination-containers which must appear in some input-set in order to get the correct answer will eventually do so.

3.3.2 Function-Generator Element Details

The function-generator element contains a prototype function, in the form of a syntax-tree. It takes FEIs out of the FEI pool and places a copy of the syntax tree into each. The local store that the coordination element created holds all the local-variables referred to in the function. Finally, the function-generator element creates an output-bag to hold outputs, then animates the reduction of the function's AST (Abstract Syntax Tree).

As the function's AST reduces, result-values get placed into the output-bag.

All actual computation which advances the program takes place by animating these Function Execution Instances. <>

Because no ordering is specified for creation of FEIs nor for the animation of FEIs, any scheduling strategy may be used. Correct results are independent of the order of creating FEIs, and independent of the order of animating FEIs (as long as the compound-constraint in the coordination element was satisfied).

3.3.3 Output Element Details

An output element takes the sets of output values generated by the animation of FEIs. It then distributes the values to the inputs connected to its unit's outputs.

The output-element implements the wires in the circuit. Each output-element is created with a "handle" to each input-pool it can place data into. Each wire coming out of a function-unit states a connection to a specific input-pool. The function-element pairs an index with each value in the results-bag. The index picks which wire that value should go on. Thus, the index picks which input-pool the value ends up in.

The output-element simply takes each value in an output-bag. It looks at the index that value is at. It looks up, in its context, the input-pool attached at that index. Then it places the value into that input-pool.

Note that no order of processing output-bags is specified, nor is any order of distributing the values in a given output-bag specified. Any causal order is valid.

3.4 Computation Sequence

The basic sequence of computation is:

→ Coordination-containers land in input pools inside a coordination element.

→ The coordination-element performs, as one atomic operation:

- Chooses an input-set which satisfies the compound-constraint
- Creates an empty Function Execution Instance (FEI)
- Creates an empty local-store in the new FEI
- Places the input-set into the new local-store
- Places the partially-complete FEI into the function-element's FEI Pool (FEIP)

→ The function-element performs, as one atomic operation:

- Chooses an FEI from the FEIP
- places a copy of the function's AST into that FEI
- creates an empty output-bag in that FEI
- reduces (executes) the function's AST. The contents of the output-bag and the contents of the containers reachable from those in the input-set will be modified as the reduction proceeds.
- places the resulting output-bag into the output-element's output-bag pool
- deletes the local-store and deletes the container which held the copy of the syntax-string which was just reduced.

→ The output-element performs, as one atomic operation:

- Takes an output-bag out of the output-pool.
- Takes each value from the output-bag, noting the index that value was at
- For a given value, looks up the input pool attached at that value's index
- places the value into that input pool
- when all values have been removed from the output-bag, deletes it

4 Meaning of Notation

4.1 Why New Notation is Used

This computation model introduces concepts which are not representable with the notation used for standard languages.

In particular, notation is needed for the following concepts:

- A store which contains other stores, called the Universal Store
- A means of addressing stores within the Universal Store
- A means of matching variables to segments of such an address
- A means of representing a circuit-element, which does not reduce, yet fits into the framework of reduction rules and syntax-directed semantics
- A means of distinguising between a store itself, the address of a store, a variable which instantiates to a store, and a variable which instantiates to an address of a store
- For a variable which instantiates to an address of a store, a means to indicate whether the address itself is meant, or the store retrieved from that address
- The reduction rules for the circuit elements must have an imperative nature. That is, the rules have an order in which the antecedents must be evaluated. This implies that the animator has state which it carries from one antecedent to the next, within a single rule
- A means to instantiate variables which appear as a portion of a larger variable name
- A means to indicate potential modification via side-effect, from one part of a rule to another.

4.1.1 Variable substitution and instantiation

In "$\sigma_{\mathrm{IP}j}$", two levels of instantiation are taking place. First the "j" is replaced in the syntax-string of the variable *name* then, that name is instantiated to a value. The name "$\sigma_{\mathrm{IP}1}$" may be used elsewhere and refers to the value that was instantiated when j had value 1.

4.1.2 The Universal Store

In code time, multiple syntax-strings exist. Some are program-syntax which is reduced by rules, some are a sequence of name-value pairs which represent a store. Some stores hold values being passed from one rule to another, some hold working data that the computation is transforming.

Thus, some entity must be specifiable which holds all these different stores. This entity is a store of stores, and is called the Universal Store, or Uber Store.

Something called an association is the address of a store in the universal store. An association has the form: $U::\mathrm{Fn}::\mathrm{myFunc}::\mathrm{Inst8}::\mathrm{FEIP}::\mathrm{ID3}::\mathrm{OutputStore}$

This form has fields separated by :: Thus the universal store is a tree structure, with each field specifying a particular branch of the tree.

4.1.3 Stores

σ represents a store. However, the symbol has a dual-nature. It instantiates to an association. However, where σ appears in a rule it means the syntax-string of name-value pairs, which results from looking up the association in $\mathbb{U}$. The special form $[\![\sigma]\!]$ means the association that the symbol "σ" instantiated(s) to.

$\mathbb{a}$ represents and association. Therfore, it also instantiates to an association. However, $\mathbb{a}$ may appear as a (reduction-rule) variable which gets replaced by its instantiated value in the syntax-string being reduced, and may instantiate from a section of the syntax string being reduced.

Because σ represents the store associated to, it cannot be used in reference to anything in a syntax-string being reduced. It is purely an object for use by the animator itself. Where an association needs to be represented or matched to in a piece of syntax-string being reduced, $\mathbb{a}$ is used.

Some examples involving associations and the universal-store:

$\mathbb{U}[\![U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{??\}::\text{OutputStore}]\!]$ means all associations which exist in the universal store, represented by $\mathbb{U}$, that match the pattern inside the black-board-braces. The $\mathbb{f}$ was previously instantiated to the name of a function. The $\mathbb{i}$ was instantiated to the number of an instance of that function. However any number following ID will match. The other terms must match exactly. Thus, this expression means the set of associations to all output stores of all IDs of the instance $\mathbb{i}$ of function $\mathbb{f}$.

In order to explicitly affect the instantiated value of σ, or retrieve the instantiated value, σ must be placed inside special brackets, like this:

$[\![\sigma]\!]$

To instantiate σ to a particular association, do this:

$[\![\sigma]\!]=U::\text{Fn}::\{\mathbb{f}\}::\text{inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{\mathbb{id}\}::\text{OutputStore}$

To instantiate an $\mathbb{a}$ to the instantiation value of σ, also an association, do this:

$\mathbb{a}=[\![\sigma]\!]$

Actually, this expression can have several meanings: If both σ and $\mathbb{a}$ were previously instantiated and have different values then this expression is false. If neither is instantiated it is likewise false. If both were instantiated and have the same value then this expression is true. If only one was previously instantiated then the other is instantiated to the same value and the expression is true.

Three equivalent ways of specifying a store (assuming $\mathbb{a}=[\![\sigma]\!]$ is true):

$\sigma \Leftrightarrow \mathbb{U}([\![\sigma]\!]) \Leftrightarrow \mathbb{U}(\mathbb{a})$

To remove the string associated to the instantiated value of σ do this:

deleteStore $\mathbb{U}([\![\sigma]\!])$ this form is used instead of just σ by convention, as a reminder of the side-effect on the universal store.

4.1.4 The Form of the Syntax-Strings That Embody Stores in the Universal Store

$''U::\text{Fn}::\text{MyFunc}::\text{Inst3}::\text{FEIP}::\text{ID42}::\text{OutputStore}|[\text{out1} := 32|\text{loopOut} := U::\text{WorkingStores}::\text{WS921}]|''$

This shows what a particular output-store looks like. The first part is the association used to retrieve this store from the universal store. The second part, between |[and]| is the store itself, consisting of name-value pairs. Each pair has name, then := then the value. The pairs are separated by |

4.1.5 Literal strings

The term ″numOutputs″ appearing in a rule is a literal string.

4.1.6 Wild Cards in Store-Names

$\mathbb{U}(U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{??\}::\text{OutputStore})$ returns every output store (in the form of a syntax-string) that has the name prifix "U: : Fn: : {$\mathbb{f}$}: : Inst{$\mathbb{i}$}: : FEIP: : ID" plus ends with "::OutputStore". In other words, the ?? inside the curly braces matches to anything.

4.1.7 Matching variables via Wild Cards in Store-Names

In $\sigma_O \in \mathbb{U}(U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\mathbb{id}?\}::\text{OutputStore})$ the $\mathbb{U}(U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\mathbb{id}?\}::\text{OutputStore})$ resolves to a set of all stores whose associations match the pattern. From this set, one is chosen and σ_O is instantiated to the association to it. At the same time, the $\mathbb{id}$ is instantiated to whatever comes between $U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}$ and $::\text{OutputStore}$ in the association instantiated to σ_O

4.1.8 Matching to Functions

$\mathbb{f}(x_1, ..., x_n)$ below the line of a rule will match to any function-name followed by "(" followed by a "," separated list of names, followed by ")" The function-name instantiates to $\mathbb{f}$ each x_i instantiates to a parameter-name, and n instantiates to the number of parameters. Thus, a function with any number of parameters will match to this form.

4.1.9 The Meaning of ⇳

The symbol ⇳ has the same semantics as ⇓ except that the RHS must always be an exact copy of the original syntax, and the rule is applied again to this result ad infinitum. It also means side-effects happen atomically, via a transaction. If any part of the rule or any sub-rules fail, all side-effects are rolled back. The side-effects update to the universal store in one atomic update.

The ordering of multiple animators reducing such rules "simultaneously" cannot be predicted. However, the results post in a definite order. All antecedents hold in the ordering which actually occurs.

Finally, rules of this type have side-effects within the rule itself. The order of the antecedents matters. The antecedents are "evaluated" top-down and left-to-right.

Because the left and right sides of ⇳ are always the same, and the rules are defined as "the antecedents hold in the order of post which actually occurs", then as many copies of the same ⇳ rule as can post and have their antecedents remain valid can be simultaneously animated with separate animators.

In essentially all useful sets of rules, the requirement that antecedents are valid in the order of actual posting, can be satisfied by choosing mutually-exclusive sets of data for each rule to "consume".

In the case of these rules, for CodeTime Circuits, simply choosing mutually-exclusive sets of input-pool contents, one valid input-set for each coord-element rule, will satisfy this condition.

4.1.10 The Meaning of ∗

$\mathbb{U}*$ represents the same syntax-string as $\mathbb{U}$ except that it may have been modified by some preceeding portion of the rule. Preceeding means that it comes previously in the top-down, left-to-right order of evaluating the "antecedents" of the rule. The $*$ is used to state explicitly that the modification happens by side-effect (if, indeed any modification does happen), and is generated in other parts of the rule or in sub-rules higher in the derivation tree which preceed the location of the $*$. The $*$ designator may also be applied to σ.

5 The Circuit Element Rules

5.1 Coordination Circuit Element

(note about $\mathbb{f}$, $\mathbb{i}$, and n: they are instantiated via matching the pattern under the line..)

generateUniqueSymbolInDomain $\mathbb{U}[\![U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\text{Unique?}\}]\!] \Downarrow \mathbb{id}$ (animator primitive)
generateAnotherStore $\mathbb{U}(U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{\mathbb{id}\}::\text{LocalStore}) \Downarrow [\![\sigma_{\text{ls}}]\!]$

$\forall j \in [1...n].\big(\ \sigma_{\text{ls}}(x_j) = \sigma_{\text{IP}\,j}(\alpha_{\text{IP}\,j}) \mid (\ \sigma_{\text{IP}\,j} \in \mathbb{U}(U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{IP}\{??\}) \wedge$
$\sigma_{\text{IP}\,j} \in \sigma_{\text{inst}}(''\text{InputPoolsDraw}'' + j + ''\text{From}'') \wedge$
$\alpha_{\text{IP}\,j} \in \text{addressesIn}\ \sigma_{\text{IP}\,j} \wedge$
$(\sigma_{\text{IP}\,j}, \alpha_{\text{IP}\,j}) \notin \boldsymbol{A}_{\text{IP}}\ )\big)$ where $\boldsymbol{A}_{\text{IP}}$ is the set of all input-pool, addr pairs which have been assigned to a local store before the point at which this atomic reduction posts

$\forall i \in [1...n].(\ \mathbb{a}_i = \sigma_{\text{ls}}(x_i)\ \wedge\ [\![\sigma_i]\!] = \mathbb{a}_i\ )$
$A = \sigma_{\text{inst}}(''\text{CoordAssertion}'')$
$\sigma_1, ..., \sigma_n \vDash A$

$\langle\ ''\text{CoordElementOf}''\,\mathbb{f}(x_1, ..., x_n)\ ''\text{instance}''\,\mathbb{i}, \mathbb{U}\rangle \Updownarrow \langle\ ''\text{CoordElementOf}''\,\mathbb{f}(x_1, ..., x_n)\ ''\text{instance}''\,\mathbb{i}, \mathbb{U}*\rangle$

5.2 Function Circuit Element

$[\![\sigma_{\text{ls}}]\!] \in \mathbb{U}[\![U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\mathbb{id}?\}::\text{LocalStore}]\!]$ (σ_{ls} is chosen & stays same for rest of rule)
(note about $\mathbb{id}$: the instantiated value of σ_{ls} will have the same value of ID in its ID field as what $\mathbb{id}$ instantiates to)
$[\![\sigma_{\text{inst}}]\!] = \mathbb{U}[\![U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{InstanceContext}]\!]$
$\forall i \in [1...n].(\mathbb{a}_i = \sigma_{\text{ls}}(x_i)\ \wedge\ [\![\sigma_i]\!] = \mathbb{a}_i)$ (means instantiate the $\mathbb{a}_i$ from the local store, then the σ_i from $\mathbb{a}_i$)
$A = \sigma_{\text{inst}}(''\text{CoordAssertion}'')$
$\sigma_1, ..., \sigma_n \vDash A$

generateEmptyStore $\mathbb{U}(U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{\mathbb{id}\}::\text{OutputStore}) \Downarrow [\![\sigma_O]\!]$
$m = \sigma_{\text{inst}}(''\text{numOutputs}'')$
addLoc$(''\text{out}'' + i)$ to $\sigma_O \mid \forall i \in [1...m]$ (create a location for each output... just add name, has empty value)

$\langle\ ''\text{FuncRules}'' \looparrowright \sigma_{\text{inst}}(''\text{FuncBody}''), [\mathbb{f}, \mathbb{i}, \mathbb{id}], [\sigma_{\text{ls}}, \sigma_{\text{ls}}, \sigma_O], \mathbb{U}\rangle \Downarrow \langle\ \mathit{null}, [\,], \mathbb{U}*\rangle$

moveStoreFromTo $(U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{\mathbb{id}\}::\text{OutputStore},\ U::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{OP}::\text{ID}\{\mathbb{id}\})$
deleteStore $\mathbb{U}([\![\sigma_{\text{ls}}]\!])$

$\langle\ ''\text{FunctionElementOf}''\,\mathbb{f}(x_1, ..., x_n)\ ''\text{instance}''\,\mathbb{i}, \mathbb{U}\rangle \Updownarrow \langle\ ''\text{FunctionElementOf}''\,\mathbb{f}(x_1, ..., x_n)\ ''\text{instance}''\,\mathbb{i}, \mathbb{U}*\rangle$

5.2.1 Context-Switch Rule

prim $\looparrowright$:{The animator saves its internal state, makes a temporary store, places the syntax-string into that store, then animates the RuleSet rules on the contents of the temporary store. It continues to perform the reductions until it reaches the form resultSyntaxString. [storeVars] is the comma separated list of sigmas, in the same order as they appear in the rules in RuleSet. In other words, once the temporary store has been made and synatxString copied into it, remove the [ ] and remove the "RuleSet $\looparrowright$" and proceed with matching RuleSet rules. When done, restore the saved animator internal state, if one desires to resume animating the circuit-element rules with that animator.}

$\langle \text{RuleSet} \looparrowright \text{syntaxString}, [\text{storeVars}], \mathbb{U}\rangle \Downarrow \langle\ \text{resultSyntaxString}, [\text{resultStoreVars}], \mathbb{U}*\rangle$

A note about the temporary store: these are created so that multiple copies of the same function-syntax-string can be animated (reduced) in overlapping fashion without interference.

5.3 Output Circuit Element

$[\![\sigma_O]\!] \in \mathbb{U}[\![U::\mathrm{Fn}::\{\mathbb{f}\}::\mathrm{Inst}\{\mathbb{i}\}::\mathrm{OP}::\mathrm{ID}\{?\mathbb{id}?\}]\!]$
(note about $\mathbb{id}$: by placement, $\mathbb{id}$ instantiates to the same value as is in the ID position of the σ_O chosen)
<sigma>$[\![\sigma_{\mathrm{inst}}]\!] = \mathbb{U}[\![U::\mathrm{Fn}::\{\mathbb{f}\}::\mathrm{Inst}\{\mathbb{i}\}::\mathrm{InstanceContext}]\!]$
$m = \sigma_{\mathrm{inst}}(''\mathrm{numOutputs}'')$
$\forall i \in [1...m].([\![\sigma_{\mathrm{IP}\,\mathbb{i}}]\!] = \mathbb{a}_i \mid \mathbb{a}_i = \sigma_{\mathrm{inst}}(''\mathrm{out}'' + i) \ \wedge \ \mathbb{a}_i \neq \mathrm{null}\ )$ ($\sigma_{\mathrm{IP}\,i}$ is null otherwise)
$\forall j \in [1...m].(\,\sigma_{\mathrm{IP}\,j}(a_j) = \mathbb{a}_j \mid \sigma_{\mathrm{IP}\,j} \neq \mathrm{null} \ \wedge \ \mathbb{a}_j = \sigma_O(''\mathrm{out}'' + j) \ \wedge \ \mathbb{a}_j \neq \mathrm{null} \ \wedge \ a_j = \mathrm{generateAddrIn}(\sigma_{\mathrm{IP}\,j})\,)$
deleteStore $\mathbb{U}([\![\sigma_O]\!])$

$\langle\, ''\mathrm{OutputElementOf}''\, \mathbb{f}(x_1, ..., x_n)\, ''\mathrm{instance}''\, \mathbb{i}, \mathbb{U} \rangle \Updownarrow \langle\, ''\mathrm{OutputElementOf}''\, \mathbb{f}(x_1, ..., x_n)\, ''\mathrm{instance}''\, \mathbb{i}, \mathbb{U} * \rangle$

6 Syntax for FuncRules

Primitive Datums:

1. integer (precision set elsewhere)
2. floating point number (precision set elsewhere)
3. symbol (only created by constant, only appears in assign and boolean)
4. character (a string is an array container holding characters)
5. assocation to a container. Containums:
   - struc container
   - array container
   - primitive container
   - system container
   - ckt-specific container

A container itself is not a data-type. Containers hold data, they are not themselves data. However, an association, itself, is data.

Primitive container is syntactic sugar for struc container where the struc specifies a single element of a given primitive type. The array container is a special kind of structure which has a dynamic number of elements, all elements of the same primitive type, and the specifier of a field of the structure is visible and can be carried in a container, as an integer. ckt-specific container is a place-holder, included it for just in case.

6.1 The meaning of value names:

Primitive Datums:
$\mathbb{a}$ is an association
n is an integer
fp is a floating point number
s is a symbol
ch is a character
p is one of any primitive datum *except* association

fd is a field of a struc container
σ is a container
σ^p is a primitive container
σ^s is a struc container
σ^a is an array container (Q: make first position be indexed by 1 or 0?)
σ^{sys} is a system container
σ^{cs} is a circuit-specific container
T^p is a primitive type
T^s is a structure type
T^a is an array type
T^{sys} is a system type
T^{cs} is a circuit-specific type
$\mathbb{Z}$ is the set of integers
$\mathbb{T}_S$ is the set of all structure-types
$\mathbb{F}_{\mathbb{T}}$ is the set of sets of fields of struc types. $\forall t \in \mathbb{T}_S$ then $F_t \in \mathbb{F}_{\mathbb{T}} = \text{set } \forall f \in \text{fields of } t$
$\mathbb{O}$ is the set of sets of outputs of functions. $\forall fn \in \mathbb{F}\mathbb{n}$ then $O_{fn} \in \mathbb{O} = \text{set } \forall o \in \text{outputs of } fn$
$\mathbb{S}$ is the set of all symbols used in the program, of the form <symbol-string>
$\mathbb{C}$ is the set of all characters
$\mathbb{A}$ is the set of all associations, of the form U::field-string $*$

6.2 General Expressions

$e ::=$	n	for $n \in \mathbb{Z}$ which is the set of all integers
$\mid$	s	for $s \in \mathbb{S}$ which is the set of all symbols
$\mid$	ch	for $ch \in \mathbb{C}$ which is the set of all characters
$\mid$	$\mathbb{a}$	for $\mathbb{a} \in \mathbb{A}$ which is the set of all associations
$\mid$	x	for $x \in L$
$\mid$	$e_1.e_2$	for $e_1, e_2 \in$ Aexp
$\mid$	$\mathbb{a}.e$	for $\mathbb{a} \in \mathbb{A}^s$, $e \in$ Aexp
$\mid$	$\mathbb{a}.fd$	for $\mathbb{a} \in \mathbb{A}^s$, $T = \text{TypeOf}[\mathbb{U}(\mathbb{a})]$, $fd \in \text{fieldsOf}[T]$
$\mid$	$e_1[e_2]$	for $e_1, e_2 \in$ Aexp
$\mid$	$\mathbb{a}[e]$	for $\mathbb{a} \in \mathbb{A}^a$, $e \in$ Aexp
$\mid$	$\mathbb{a}[n]$	for $\mathbb{a} \in \mathbb{A}^a$, $n \in [1, 2, ...]$
$\mid$	o	for $o \in \text{addressesIn}[\sigma_O]$
$\mid$	$e_1 \boxtimes e_2$	for $e_1, e_2 \in$ Aexp, $\boxtimes \in$ arity 2 arithmetic operators
$\mid$	$\boxslash e_1$	for $e_1 \in$ Aexp, $\boxslash \in$ arity 1 arithmetic operators

6.3 Boolean Expressions

$b ::=$	true	
$\mid$	false	
$\mid$	$e_1 == e_2$	$e_1, e_2 \in$ Aexp
$\mid$	$e_1 < e_2$	$e_1, e_2 \in$ Aexp
$\mid$	$e_1 <= e_2$	$e_1, e_2 \in$ Aexp
$\mid$	$e_1 > e_2$	$e_1, e_2 \in$ Aexp
$\mid$	$e_1 >= e_2$	$e_1, e_2 \in$ Aexp
$\mid$	$\neg b$	$b \in$ Bexp
$\mid$	$b_1 \wedge b_2$	$b_1, b_2 \in$ Bexp
$\mid$	$b_1 \vee b_2$	$b_1, b_2 \in$ Bexp

6.4 Commands

$c ::=$	Pml= new t	for $t \in \text{namesOf}[\mathbb{T}_S]$, $P \in [\text{null}, k, d]$
$\mid$	PmT=OTC	for $P \in [\text{null}, d, k], T \in [l, c], O \in [t, c], C \in [\text{null}, c]$
$\mid$	output e_1 to e_2 withTag $\{c\}$	for $e_1, e_2 \in \text{Aexp}, c \in \text{Comm}$
$\mid$	c_1 ; c_2	$c_1, c_2 \in \text{Comm}$
$\mid$	if b then $\{c_1\}$ else $\{c_2\}$	for $c_1, c_2 \in \text{Comm}, b \in \text{Bexp}$
$\mid$	$\mathbb{a}_1.fd_1 \;\; m = t \;\; \mathbb{a}_2.fd_2$ and etc...	for $\mathbb{a}_1, \mathbb{a}_2 \in \mathbb{A}$, $T_1 = \text{TypeOf}[\mathbb{U}(\mathbb{a}_1)], T_2 = \text{TypeOf}[\mathbb{U}(\mathbb{a}_2)]$ $fd_1 \in \text{fieldsOf}[T_1], fd_2 \in \text{fieldsOf}[T_2]$
$\mid$	endFn	

The set of all the types, $\boldsymbol{T}_s$, contains name value pairs. The name is pre-fixed by the position in the hierarchy in the *source* code, or something equivalent, to distinguish two types that have the same name. IE two different versions of "myStruc" appear in different hierarchy locations in the source code. So, they're each pre-fixed by the names of the hierarchy location, starting from root.

7 Function Reduction Rules

These are the "FuncRules"

7.1 New rule

$$\frac{T \in \boldsymbol{T}_s \quad \left\langle \text{generateAnotherStore}\left[\mathbb{U}(U::\text{WorkingStores}::\{?\text{Unique}?\}), T\right], \mathbb{U} \right\rangle \Downarrow \left\langle \mathbb{a}, \mathbb{U} * \right\rangle \quad \text{(animator primitive)}}{\langle \text{new } T, \mathbb{U} \rangle \Downarrow \langle \mathbb{a}, \mathbb{U} * \rangle}$$

The "new" command can only appear on the RHS of an assignment command. It cannot appear in combination with any other expressions or commands.

7.2 Select rules

A source-code example of a selection is "T.myInt". Here, "T" is a local variable, which means that it is an address in σ_{ls}, the local-store. The address's paired value is an association to a store that, in turn, has an addr "myInt". So, T resolves to an association, then myInt is recognized as an addr in the associated store, and this expr reduces to "$\mathbb{a}.fd$" which is then either used in an assignment as-is, or, if in an arithmetic operation or other op that wants the value, it is reduced to an int. The reduction happens by looking up the value in the location.

In the source-code expression "T.myArray[5]" "T" is again a local variable which is an address in σ_{ls}. The address's paired value is an association to a store that, in turns, has an address "myArray". The contents of "myArray" addr is an association to an array container. This expr is reduced to "a.myArray[5]" then to "a.*fd* [5]" then the contents of the location is retrieved to give "$\mathbb{a}_1[5]$" and finally, the index is used to give whatever is the contents of the array position whose name is the symbol "5".

7.2.1 Var to association

Ex: "T"

$$\frac{fd = x \qquad \text{Type}[\sigma_{\text{ls}}] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd \in \text{fieldsOf}[T_1] \qquad \mathbb{a}_2 = \sigma_{\text{ls}}(fd) \qquad [\![\sigma_z]\!] = \mathbb{a}_2}{\langle x, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathbb{a}_2, \sigma_z, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.2 Var to location

Ex: "T"

$$\frac{fd = x \qquad \text{Type}[\sigma_{\text{ls}}] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd \in \text{fieldsOf}[T_1] \qquad \mathbb{a} = [\![\sigma_{\text{ls}}]\!]}{\langle x, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathbb{a}.fd, \sigma_{\text{ls}}, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.3 Expr.Expr to Expr.Expr

$$\frac{\langle e_1, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle e_1', \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \quad \langle e_2, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle e_2', \sigma_z, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}{\langle e_1.e_2, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle e_1'.e_2', \sigma_z, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

Order matters. Want sigma to change due to e_1, then e_2 to be reduced given that new sigma. The reduction of e_2 possibly changes sigmas again. Don't want e_2 to be reduced first, change sigma, and *then* have e_1 reduced with that sigma. Thus, have a single rule which states this order. If e_1 or e_2 is already reduced then e_1 must be in the form of an association, and e_2 must be in the form of a field in the type of the associated store. In either case, a different rule will match. This rule is only for the case that both expressions reduce to simpler expressions.

7.2.4 Expr to association

$$\frac{\mathbb{a} = e \qquad \mathbb{a} \in \mathbb{A}}{\langle e, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathbb{a}, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.5 Expr.Expr to Assoc.Expr

$$\frac{\langle e_1, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathbb{a}, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}{\langle e_1.e_2, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathbb{a}.e_2, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.6 Expr.Expr to Expr.field

$$\frac{\langle e_2, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle fd, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}{\langle e_1.e_2, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle e_1.fd, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

Not sure if this rule can ever be used. It may be that an expression cannot reduce to a field before the association has reduced.

7.2.7 Assoc.Expr to location

$$\frac{\text{Type}[\mathbb{U}(\mathfrak{a})] = T \qquad T \in \boldsymbol{T}_s \qquad fd \in \text{fieldsOf}[T]}{\langle \mathfrak{a}.e_1, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathfrak{a}.fd, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.8 Loc to association

Ex: ".myArray"

$$\frac{\text{Type}[\mathbb{U}(\mathfrak{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd \in \text{fieldsOf}[T_1] \qquad [\![\sigma_x]\!] = \mathfrak{a}_1 \qquad \mathfrak{a}_2 = \sigma_x(fd) \qquad [\![\sigma_y]\!] = \mathfrak{a}_2}{\langle \mathfrak{a}_1.fd, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathfrak{a}_2, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.9 Loc to value

Ex: ".myInt"

$$\frac{\text{Type}[\mathbb{U}(\mathfrak{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd \in \text{fieldsOf}[T_1] \qquad [\![\sigma_x]\!] = \mathfrak{a}_1 \qquad p = \sigma_x(fd)}{\langle \mathfrak{a}_1.fd, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle p, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.10 expr [expr] to array [expr]

$$\frac{\langle e_1, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathfrak{a}, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \qquad \text{Type}[\mathbb{U}(\mathfrak{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_{\text{arrays}}}{\langle e_1[e_2], \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathfrak{a}[e_2], \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.11 Array [expr] to array [index]

$$\frac{\begin{array}{l}\text{Type}[\mathbb{U}(\mathfrak{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_{\text{arrays}} \\ \langle e, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle n, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \qquad n_2 \in \text{fieldsOf}[T_1]\end{array}}{\langle \mathfrak{a}[e], \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathfrak{a}[n], \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.12 Array [index] to value

Ex: "[5]" if type that is retrieved is *not* an association:

$$\frac{\text{Type}[\mathbb{U}(\mathfrak{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_{\text{arrays}} \qquad n \in \text{fieldsOf}[T_1] \qquad [\![\sigma_x]\!] = \mathfrak{a}_1 \qquad p = \sigma_x(n)}{\langle \mathfrak{a}_1[n], \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle p, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.13 Array [index] to association

Ex: "[5]" if type that is retrieved is an association:

$$\frac{\text{Type}[\mathbb{U}(\mathfrak{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_{\text{arrays}} \qquad n \in \text{fieldsOf}[T_1] \qquad \mathfrak{a}_2 = \sigma_x(n_2) \qquad [\![\sigma_y]\!] = \mathfrak{a}_2}{\langle \mathfrak{a}_1[n], \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \mathfrak{a}_2, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle}$$

7.2.14 Arithmetic rules (just one example to show form)

$$\frac{\langle e_1, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow n_1 \qquad \langle e_2, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow n_2 \qquad n \text{ is primitive sum of } n_1 \text{ and } n_2}{\langle e_1 + e_2, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow n}$$

Notice the σ_x in this rule. Either of the expressions above the line may end up changing the sigma via a selection operation within the expression, then resolve to an integer in that different sigma. However, expressions are only able to resolve to the equivalent of reads. It is only possible to match to the form "e" if there are no commands inside. (ie, no "new" no "assign", no "output" only selectors and arithmetic). Thus, despite possible changes inside the expressions, σ_x remains the same from place to place in this rule.

7.3 Container management

Four possible commands may be valuable to have in functions for managing containers (stores).

DeleteContainer

NewContainer

DeleteContents Is library code which first deletes, if want old associated containers deleted, then assigns contents to null.

SetAllLocations A library routine which sets all locations in a container to a given value. If want old associated-to containers deleted, first deletes them.

7.4 Assignment

Both left-hand side and right-hand side have behaviors. These behaviors are the result of side-effects. The side effects are creation and deletion of containers, and modification of contents.

The same expression in a source language can be treated either as an association or as a location, so the form of assignment must clarify. For example:

"T.myContainer = In.aContainer;" does this mean copy the *association* paired to the "aContainer" address, or does it mean copy the *contents* of the container associated to by the value paired to the "aContainer" address. What about side-effects? if an association is copied, that means two separate associations to the same storage exist, which enables side-effects.

Likewise, the LHS has an association to a container before the assignment operation takes place. The container associated to by the LHS before the assignment may have other associations to it, if side-effects are allowed (they are in extensions of this base spec). In that case, should the other containers have associations continue to see that container, or should this assignment delete it, causing those other associations to become dangling? The assignment operation must specify which behavior to exhibit.

The way the behavior is specified is by stating both the type of specifier each side of the assignment should reduce to, and what operation should be performed on the thing specified. In other words, state whether each side should reduce to a location holding a value, a location holding an association, or to just an association, plus an operation to apply to that thing.

Here are the allowed types that the side's expr reduces to, and the operations to apply to it:

LHS:

- type that expr reduces to: loc(value), loc(assoc), assoc
- behavior implied by type: change value, change assoc in loc, change contents of container assoc to
- effect on pre-assignment containers: keep container(s) at other end of association(s), delete container(s) at other end of association(s). IE, if LHS reduces to an association, and the associated-to container has locations which hold associations, then those 2nd-level containers will be either kept or deleted. If a container associated to is deleted, it may have had associations in it. Deleting that container may orphan ones associated to by it. Garbage collection detects and deletes these orphaned containers. If multiple associations exist to some of those 2nd or lower level containers, then they are not orphans and will not be garbage collected. The programmer may go down into the lower level containers explicitly and delete containers that have multiple associations that they still wish to be deleted. (Note on implementation: these semantics allow the VS-compiler to determine which containers need to be considered for garbage collection and which not, and also tells VS-compiler which locations need to have null-pointer checks placed on them. A higher-level type checker should be able to eliminate most of the tedium from source languages.)

RHS:

- type that expr reduces to: loc(value), loc(assoc), assoc, temp value, temp assoc
- operation specified: copy loc(v), copy then delete contents loc(v), copy cont loc(a), copy then delete cont loc(a), copy container assoc to by $\mathbb{a}$, copy then delete contents of each location in container assoc to by $\mathbb{a}$.
- implied operations: temp value and temp association both copy then delete (transfer) by default.
- effect on pre-assignment containers: if copy operation, then none. If transfer (copy then delete), then whatever the RHS expr is interpreted as is deleted after the assignment operation. Delete something on RHS means that location(s) become empty. For example, if the operation was transfer contents of container, then the RHS container remains, but each location is empty. There are no deeper level effects because all the associations still exist in their original form and without additional copies (the associations themselves are just in a different location now). No recursive forms are specified because they can be provided by library routines.

A note on notation: in all rules except assignment, locations can be indicated simply by a field name. The container that field exists in is the first sigma. However, assignment may have a location on each side of the = sign. Thus, it cannot use this form. Instead, "$\mathbb{a}.fd$" is used on both sides. One alternative would be to use an embedded notation like this:

"$\langle\langle fd_1, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle$ kml = ta $\langle fd_2\,, \sigma_y, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle$" which has a sigma on each side of the = sign. This is a nice notation, might use it in future.

7.5 The Assignment Rules:

Various combinations of letters determine what each side should resolve to, and what operation to apply to each side.

A note about new and temporary associations: New can only be used alone on the RHS of an assignment. It cannot appear in combination with any other commands or expressions. It is the only means to get a temporary association on the RHS, and always produces an empty container.

1. loc(v) **ml**= temp value "T.myInt ml= 5;"

   → the temporary value is placed into the location.. don't need RHS designator

   $$\begin{array}{l}\text{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \text{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \sigma_1(fd_1) = p \\ \hline \langle \mathbb{a}_1.fd_1 \;\text{ml}=\; p\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle\end{array} \quad \rightarrow$$

2. loc(a) **dml**= temp assoc "T.myContainer dml= new Container;"

   → the temporary association is placed into the LHS location.. don't need RHS designator.. delete the container associated to by previous contents of LHS..

   $$\begin{array}{l}\text{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_{s,a} \qquad fd_1 \in \text{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \text{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \\ \text{Type}[\sigma_1(fd_1)] = \mathbb{a}\colon T_{11} \qquad\qquad T_2 = T_{11} \\ \big(\,\text{delete}[\mathbb{U}(\,\sigma_1(fd_1))]\;\big|\;\text{Type}[\sigma_1(fd_1)] \in \mathbb{a}\colon \boldsymbol{T}_{s,a}\,\big) \\ \sigma_1(fd_1) = \mathbb{a}_2 \\ \hline \langle \mathbb{a}_1.fd_1 \;\text{dml}=\; \mathbb{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle\end{array} \quad \rightarrow$$

3. loc(a) **kml=** temp assoc "T.myContainer kml= new Container;"

   → the temporary association is placed into the LHS location.. keep in existence the container associated to by previous contents of LHS..

$$\begin{array}{l}\mathrm{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_{s,a} \qquad fd_1 \in \mathrm{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \mathrm{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \\ \rightarrow \quad \mathrm{Type}[\sigma_1(fd_1)] = \mathbb{a}\colon T_{11} \qquad\qquad T_2 = T_{11} \\ \sigma_1(fd_1) = \mathbb{a}_2 \\ \hline \langle \mathbb{a}_1.fd_1 \ \mathrm{kml} = \ \mathbb{a}_2, \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} * \rangle \end{array}$$

   note about σ_x: it will be either σ_1 or σ_2 the alternative rule form will fix this uncertainty.

4. assoc **{d,k}mc=** temp assoc "T.myContainer dmc= new Container;"

   → Disallowed because can only ever get empty, new, containers via *temp* associations. Thus, makes no sense to transfer the contents from an empty container! Delete contents is provided as a library routine.

5. loc(v) **ml=tlc** selected loc(v) "T.myInt ml=tlc Temp.anInt;"

   → transfer contents from location on RHS to location on LHS. means RHS container will have an empty spot with nothing in it after this command finishes.

$$\begin{array}{l}\mathrm{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \mathrm{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \mathrm{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad fd_2 \in \mathrm{fieldsOf}[T_2] \qquad [\![\sigma_2]\!] = \mathbb{a}_2 \\ \rightarrow \quad \sigma_1(fd_1) = \sigma_2(fd_2) \qquad \sigma_2(fd_2) = \mathit{null} \\ \hline \langle \mathbb{a}_1.fd_1 \ m = t \ \ \mathbb{a}_2.fd_2, \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} * \rangle \end{array}$$

6. loc(a) **kml=ta** loc(a) Ex: "T.myContainer kml=ta In.aContnr;"

   → The association held inside the RHS location is moved to the LHS location.

   → Also have to state what happens to all the containers which used to be associated to by the former contents of the LHS container.. either keep or delete.. in "kml=" the k means keep the previously-associated-to container around..

$$\begin{array}{l}\mathrm{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \mathrm{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \mathrm{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad fd_2 \in \mathrm{fieldsOf}[T_2] \qquad [\![\sigma_2]\!] = \mathbb{a}_2 \\ T_1 = T_2 \\ \rightarrow \quad \mathrm{Type}[\sigma_1(fd_1)] = \mathbb{a}\colon T_{11} \qquad\qquad \mathrm{Type}[\sigma_2(fd_2)] = \mathbb{a}\colon T_{22} \\ T_{11} = T_{22} \\ \sigma_1(fd_1) = \sigma_2(fd_2) \qquad \sigma_2(fd_2) = \mathit{null} \\ \hline \langle \mathbb{a}_1.fd_1 \ \mathrm{kml} = \mathrm{ta} \ \ \mathbb{a}_2.fd_2, \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} * \rangle \end{array}$$

7. loc(a) **dml=ta** loc(a) Ex: "T.myContainer dml=ta In.aContnr;"

   → The association held inside the RHS location is moved to the LHS location.

   → the d on LHS of = means delete the container previously-associated-to by the contents of the LHS location..

$$\begin{array}{l}\mathrm{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \mathrm{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \mathrm{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad fd_2 \in \mathrm{fieldsOf}[T_2] \qquad [\![\sigma_2]\!] = \mathbb{a}_2 \\ T_1 = T_2 \\ \mathrm{Type}[\sigma_1(fd_1)] = \mathbb{a}\colon T_{11} \qquad\qquad \mathrm{Type}[\sigma_2(fd_2)] = \mathbb{a}\colon T_{22} \\ \rightarrow \quad T_{11} = T_{22} \\ \big(\ \mathrm{delete}[\mathbb{U}(\ \sigma_1(fd_1))] \ \big|\ \mathrm{Type}[\sigma_1(fd_1)] \in \mathbb{a}\colon \boldsymbol{T}_{s,a}\ \big) \\ \sigma_1(fd_1) = \sigma_2(fd_2) \qquad \sigma_2(fd_2) = \mathit{null} \\ \hline \langle \mathbb{a}_1.fd_1 \ \mathrm{kml} = \mathrm{ta} \ \ \mathbb{a}_2.fd_2, \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_x, \sigma_{\mathrm{ls}}, \sigma_O, \mathbb{U} * \rangle \end{array}$$

8. assoc **kmc=tc** selected assoc Ex: "T.myContainer kmc=tc In.aContnr;"

   → The "c" on the LHS of = means treat LHS as a container and transfer the contents from RHS.. any values, including associations, that used to be inside the RHS container are no longer there after this command finishes.. the addresses will still be in the RHS container, but the values they're paired to are empty..

   → the k on LHS of = means keep in existence the containers previously-associated-to by locations in the container the LHS associates to..

   $$\begin{array}{l} \text{Type}[\mathbb{U}(\mathtt{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad [\![\sigma_1]\!] = \mathtt{a}_1 \\ \text{Type}[\mathbb{U}(\mathtt{a}_2)] = T_2 \qquad T_2 = T_1 \qquad [\![\sigma_2]\!] = \mathtt{a}_2 \\ \forall fd_1 \in \text{fieldsOf}[T_1].\,(\,\sigma_1(fd_1) = \sigma_2(fd_1)\,) \\ \forall fd_1 \in \text{fieldsOf}[T_1].\,(\,\sigma_2(fd_1) = null\,) \end{array} \rightarrow$$

   $$\langle \mathtt{a}_1 \;\; \text{kmc} = t \;\; \mathtt{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle$$

9. assoc **dmc=tc** selected assoc Ex: "T.myContainer m=tc In.aContnr;"

   → The "c" on the LHS of = means treat the LHS as a container and transfer contents from RHS container to interior of LHS container.. any values, including associations, that used to be inside the RHS container are no longer there after this command finishes.. the addresses in the RHS container will still be there, but the values they're paired to are empty..

   → State what happens to all the containers which used to be associated to by the former contents of the LHS container.. either keep or delete.. in "dmc=" the d means delete the previously-associated-to containers..

   $$\begin{array}{l} \text{Type}[\mathbb{U}(\mathtt{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad [\![\sigma_1]\!] = \mathtt{a}_1 \\ \text{Type}[\mathbb{U}(\mathtt{a}_2)] = T_2 \qquad T_2 = T_1 \qquad [\![\sigma_2]\!] = \mathtt{a}_2 \\ \forall fd_1 \in \text{fieldsOf}[T_1].\,\big(\,\text{delete}[\mathbb{U}(\,\sigma_1(fd_1))] \;\big|\; \text{Type}[\sigma_1(fd_1)] \in \mathtt{a}\colon \boldsymbol{T}_{s,a}\,\big) \\ \forall fd_1 \in \text{fieldsOf}[T_1].\,(\,\sigma_1(fd_1) = \sigma_2(fd_1)\,) \\ \forall fd_1 \in \text{fieldsOf}[T_1].\,(\,\sigma_2(fd_1) = null\,) \end{array} \rightarrow$$

   $$\langle \mathtt{a}_1 \;\; \text{dmc} = t \;\; \mathtt{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle$$

10. loc(v) **ml=cl** selected loc(v) "T.myInt m=c Temp.anInt;"

    → copy contents from location on RHS to location on LHS..

    $$\begin{array}{l} \text{Type}[\mathbb{U}(\mathtt{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \text{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathtt{a}_1 \\ \text{Type}[\mathbb{U}(\mathtt{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad fd_2 \in \text{fieldsOf}[T_2] \qquad [\![\sigma_2]\!] = \mathtt{a}_2 \\ \sigma_1(fd_1) = \sigma_2(fd_2) \end{array} \rightarrow$$

    $$\langle \mathtt{a}_1.fd_1 \;\; \text{ml} = c \;\; \mathtt{a}_2.fd_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle$$

11. **loc(a) kml=ca selected assoc** "T.myArray ml=ca Temp.tempArray;"

    → copy association from RHS to location on LHS..

    → This rule is disallowed until side effects are added

    $$\begin{array}{l} \text{Type}[\mathbb{U}(\mathtt{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \text{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathtt{a}_1 \\ \text{Type}[\sigma_1(fd_1)] = \mathtt{a}\colon T_{11} \\ \text{Type}[\mathbb{U}(\mathtt{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad T_{11} = T_2 \\ \sigma_1(fd_1) = \mathtt{a}_2 \end{array} \rightarrow$$

    $$\langle \mathtt{a}_1.fd_1 \;\; \text{kml} = \text{ca} \;\; \mathtt{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle$$

12. **loc(a) dml=ca selected assoc** "T.myArray ml=ca Temp.tempArray;"

    → copy association from RHS to location on LHS.. delete the container that used to be associated to by contents of the LHS location..

→ This rule is disallowed until side effects are added

$$\begin{array}{l}\text{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \text{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \text{Type}[\sigma_1(fd_1)] = \mathbb{a}\text{:}\, T_{11} \\ \text{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad T_{11} = T_2 \\ \rightarrow \quad \text{delete}[\mathbb{U}(\,\sigma_1(fd_1))] \mid \text{Type}[\sigma_1(fd_1)] \in \mathbb{a}\text{:}\, \boldsymbol{T}_{s,a} \\ \sigma_1(fd_1) = \mathbb{a}_2 \\ \hline \langle \mathbb{a}_1.fd_1 \;\; \text{dml} = \text{ca} \;\; \mathbb{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} *\rangle \end{array}$$

13. **loc(a) dml=cac selected assoc** "T.myArray dml=cac Temp.tempArray;"

→ copy container that RHS associates to and place association to copy into location on LHS.. Delete the container which used to be associated to by the LHS loc.

→ This is the same command as "d=cc"

$$\begin{array}{l}\text{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \text{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \text{Type}[\sigma_1(fd_1)] = \mathbb{a}\text{:}\, T_{11} \\ \text{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad T_{11} = T_2 \\ \rightarrow \quad \big(\text{delete}[\mathbb{U}(\,\sigma_1(fd_1))] \mid \text{Type}[\sigma_1(fd_1)] \in \mathbb{a}\text{:}\, \boldsymbol{T}_{s,a}\big) \\ \mathbb{a}_3 = \text{copyOf}[\mathbb{U}(\mathbb{a}_2)] \\ \sigma_1(fd_1) = \mathbb{a}_3 \\ \hline \langle \mathbb{a}_1.fd_1 \;\; \text{dml} = \text{cc} \;\; \mathbb{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} *\rangle \end{array}$$

14. **loc(a) kml=cac selected assoc** "T.myArray ml=ca Temp.tempArray;"

→ copy container that RHS associates to and place association to copy into location on LHS.. keep the container which used to be associated to by the LHS loc..

→ This is the same command as "k=cc"

$$\begin{array}{l}\text{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad fd_1 \in \text{fieldsOf}[T_1] \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \text{Type}[\sigma_1(fd_1)] = \mathbb{a}\text{:}\, T_{11} \\ \text{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 \in \boldsymbol{T}_s \qquad T_{11} = T_2 \\ \rightarrow \quad \mathbb{a}_3 = \text{copyOf}[\mathbb{U}(\mathbb{a}_2)] \\ \sigma_1(fd_1) = \mathbb{a}_3 \\ \hline \langle \mathbb{a}_1.fd_1 \;\; \text{kml} = \text{cc} \;\; \mathbb{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} *\rangle \end{array}$$

15. **assoc kmc=cc selected assoc** Ex: "T.myContainer kmc=cc In.aContnr;"

→ Have to state the way to treat LHS.. as either a container or as a location. The "c" on the LHS of = means treat the LHS as a container.

→ So, copy contents from RHS container to interior of LHS container..

→ State what happens to all the containers which used to be associated to by the former contents of the LHS container.. either keep or delete.. in "kmc=" the k means keep the previously-associated-to containers around..

→ For now, this rule is disallowed. When add side-effects, the type system will enforce stating conditions on the side-effects to keep them safe. The side-effects are created by copying associations. After the copy two separate containers hold associations to the same third container, allowing the two containers to go to separate function-units and change the data each of the two sees in the third via side-effect.

$$\begin{array}{l}\text{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \text{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 = T_1 \qquad [\![\sigma_2]\!] = \mathbb{a}_2 \\ \rightarrow \quad \forall fd_1 \in \text{fieldsOf}[T_1].\,(\,\sigma_1(fd_1) = \sigma_2(fd_1)\,) \\ \hline \langle \mathbb{a}_1 \;\; \text{kmc} = cc \;\; \mathbb{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} *\rangle \end{array}$$

16. assoc **dmc=cc** selected assoc Ex: "T.myContainer dmc=c In.aContnr;"
    - → Have to state the way to treat LHS.. as either a container or as a location. The "c" on the LHS of = means modify the contents of LHS, which implies treat RHS as a container. Could also write this as "dmc=c" with LHS "c" implying that the contents will be copied.. but putting both c s makes it clearer..
    - → So, copy contents from RHS container to interior of LHS container..
    - → State what happens to all the containers which used to be associated to by the former contents of the LHS container.. either keep or delete.. in "dmc=" the d means delete the previously-associated-to containers..
    - → For now, this rule is disallowed. When add side-effects, the type system will enforce stating conditions on the side-effects to keep them safe. The side-effects are created by copying associations. After the copy two separate containers hold associations to the same third container, allowing the two containers to go to separate function-units and change the data each of the two sees in the third via side-effect.

$$\frac{\begin{array}{l}\text{Type}[\mathbb{U}(\mathbb{a}_1)] = T_1 \qquad T_1 \in \boldsymbol{T}_s \qquad [\![\sigma_1]\!] = \mathbb{a}_1 \\ \text{Type}[\mathbb{U}(\mathbb{a}_2)] = T_2 \qquad T_2 = T_1 \qquad [\![\sigma_2]\!] = \mathbb{a}_2 \\ \forall fd_1 \in \text{fieldsOf}[T_1].\,(\,\text{delete}[\mathbb{U}(\,\sigma_1(fd_1))] \mid \text{Type}[\sigma_1(fd_1)] \in \mathbb{a}{:}\,\boldsymbol{T}_{s,a}\,) \\ \forall fd_1 \in \text{fieldsOf}[T_1].\,(\,\sigma_1(fd_1) = \sigma_2(fd_1)\,)\end{array}}{\langle \mathbb{a}_1 \;\; \text{dmc} = cc \;\; \mathbb{a}_2\,, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle}$$

7.6 If rule

$$\frac{\langle b, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \text{true} \qquad \langle c_1, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle}{\langle \text{if } b \text{ then } \{\,c_1\,\} \text{ else } \{c_2\}, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \;\Downarrow\; \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle}$$

$$\frac{\langle b, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \text{false} \qquad \langle c_2, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle}{\langle \text{if } b \text{ then } \{\,c_1\,\} \text{ else } \{c_2\}, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \;\Downarrow\; \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U} * \rangle}$$

don't need the curly braces, 'cause the rule acts on an abstract syntax tree, and curly braces are just for the parser, but put in anyway just for clarity..

7.7 Output rule

$$\frac{\begin{array}{l}o \in \text{addressesIn}[\sigma_O] \\ e \Downarrow \mathbb{a}_r \qquad \sigma_O(o) := \mathbb{a}_r \Downarrow \sigma_O[o := \mathbb{a}_r] \qquad [\![\sigma_r]\!] = \mathbb{a}_r \\ \langle\, \text{"TagRules"} \looparrowright c, [\sigma_r, \sigma_{\text{ls}}, \sigma_O], \mathbb{U}\rangle \Downarrow \langle \mathit{null}, [\sigma_r *, \sigma_{\text{ls}} *, \sigma_O *], \mathbb{U} * \rangle\end{array}}{\langle\, \text{output } e \text{ to } o \text{ withTag } \{\,c\,\}, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathbb{U}\rangle \Downarrow \langle\, \sigma_x, \sigma_{\text{ls}}, \sigma_O[o := \mathbb{a}_r], \mathbb{U} * \rangle}$$

8 Conclusion

This paper has shown the operational semantics for the circuit-format used to distribute programs in the CodeTime Parallel Software Platform. These semantics have been presented in terms of extended semantic "machinery." The new machinery allows expressing side-effects within the rules themselves, and circuit-like behavior, while still maintaining a syntax-directed framework.