The Big-Step Operational Semantics of Code Time Circuits

BY

Sean Halle

Abstract

The Code Time Parallel Programming 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 is invariant to the number of animators (processors), invariant to changing the size of data, allows simply and automatically changing the code-length of a schedulable work-unit, allows source-languages which provide easy identification and specification of parallel tasks, allows source-languages which provide simple determination and declaration of coordination of parallel tasks, and 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 Code Time 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 virtual OS remains the same across all physical platforms. The user perceives a single machine with a fixed set of commands. 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 a virtual OS. These portions of the Code Time platform are detailed elsewhere.

In this paper, the details of the distribution-format are given. The format uses a circuit paradigm to represent programs. The operational semantics are given for this format. These semantics are based upon the Big Step semantics, however, some modifications and extensions have been made to the “machinery” of Big Step semantics.

The details of this extended semantic machinery are also given.

2 Related Work

Large Grain Data Flow has a similar structure as Code Time. However, Code Time generalizes the semantics of data flow and introduces concepts to it. 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 Code Time, 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 degree.

3 The Elements of a Code Time Virtual Circuit

A virtual circuit is composed of function-units plus wires connecting the units. Each unit contains three primitive elements, a coordination-element, a function-element, and an output-element. Units are the basic building blocks of a Code Time Circuit. In this, first, version of Code Time Circuits, only function units and system units exist. The primitive elements inside the function units perform all the 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 Units

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

The first kind of unit, function-unit, performs the computation of the circuit. The inputs to the unit are wired to pools of data residing in a coordination element. The coordination element consumes the input data and generates sets of input-parameters. A function element consumes the sets of input parameters and generates a schedulable work-unit from each. Upon execution, each work-unit produces a bag of output values. An output element consumes these bags of output values and distributes them to the input pools of destination units wired to this unit’s outputs.
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 the, visible, operating system.

3.2 Primitive Circuit Elements

Three kinds of circuit element exist, in this initial version of CT circuits. These are coordination elements, function elements, and output elements.

A coordination element’s purpose is to direct the order in which computations take place. It generates input-sets. Each set must satisfy an assertion about its contents. In particular, the contents of one container relative to the contents of the other containers is checked. This assertion takes the place of the various synchronisation mechanisms used in other languages.

A function element contains a snippet of code and is like a section of a function-call in C. It takes the input-sets generated by the coordination-element and creates a schedulable work-unit from it, called a Function Execution Instance (FEI). The FEI contains the input-set, a copy of the function-unit’s code snippet, and a bag to hold outputs generated by that code snippet. All actual computation takes place inside Function Execution Instances. Some, undefined, scheduling strategy chooses when and “where” to animate each FEI. In contrast, only the order of computation is affected by coordination elements.

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.

4 Computation Sequence

The basic sequence of computation is:

→ A control-state container lands in an input pool. Input pools are inside coordination elements. A control-state container is a container which has a special field called a tag.

→ Coordination element creates input sets from the containers in the input pools. An input set is defined as a set of containers whose values satisfy a given boolean. The tag field is for use exclusively by this boolean. The Coord element determines if a combination of containers passes the test encoded by the boolean. The order in which this checking takes place is undefined. Each container (more precisely, each association to a container) which enters an input pool may only appear in a single input-set.

→ The coordination-element atomically:
   - Packages the associations to containers that passed the test, as an input-set.
   - Creates a container to hold the input-set plus all of the function’s local variables.
   - Places that local-store into the function-element’s Function Execution Instance Pool (FEIP).

→ The function-element atomically:
   - Creates an output container and a copy of the function’s syntax-string.
– At some undefined time, in some undefined order, the copy of the syntax-string is reduced (executed). The execution follows the syntax-directed semantics.

– Modifies the state of containers, as a side-effect of execution, in the function-execution-instance’s input-set, and/or creates and modifies new containers.

– Places associations into the output-container. Each “output” statement in the syntax-string places one of the modified or created containers into the output container.

– At completion of the reduction, places the output-container into the output-pool in the output-element.

– At completion of the reduction, deletes the local-store and deletes the container which held the copy of the syntax-string which has just been reduced.

→ The output-element atomically:

– in some undefined order, including possible overlap, gets an output-container out of the output-pool.

– Takes each value (which is always an association) from the output-container, looks up in the instance-context what input pool is attached to the output-name that was output to, and places the value into that input pool.

– Deletes the output container when it’s done.

5 Meaning of Notation (Rules for the Animator)

5.0.1 Variable substitution and instantiation

In “$\sigma_{IP_j}$”, 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 “$\sigma_{IP_1}$” may be used elsewhere and refers to the value that was instantiated when j had value 1.

5.0.2 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: $U::Fn::myFunc::Inst8::FEIP::ID3::OutputStore$

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

5.0.3 Stores

$\sigma$ represents a store. However, the symbol has a dual-nature. It instantiates to an association. However, where $\sigma$ appears in a rule it means the syntax-string of name-value pairs, which results from looking up the association in $\Sigma$. The special form $\exists K$ means the association that the symbol “$\sigma$” instantiated(s) to.
a represents and association. Therefore, it also instantiates to an association. However, 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, a is used.

Some examples involving associations and the universal-store:

∪Ur : : Fn : : {f} : : Inst{i} : : FEIP : : ID{??} : : OutputStore means all associations which exist in the universal store, represented by U, that match the pattern inside the black-board-braces. The f was previously instantiated to the name of a function. The 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 i of function 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:

\[ \sigma = \text{U} : : \text{Fn} : : \{f\} : : \text{Inst}\{i\} : : \text{FEIP} : : \text{ID}\{i\} : : \text{OutputStore} \]

To instantiate an a to the instantiation value of σ, also an association, do this:

\[ a = \sigma \]

Actually, this expression can have several meanings: If both σ and 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 a = \sigma \ K is true):

\[ \sigma \ Leftrightarrow \text{U}(\sigma \ K) \ Leftrightarrow \text{U}(a) \]

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

\[ \text{deleteStoreU}(\sigma \ K) \]

this form is used instead of just σ by convention, as a reminder of the side-effect on the universal store.

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


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 |.

5.0.5 Literal strings

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

5.0.6 Wild Cards in Store-Names


returns every output store (in the form of a syntax-string) that has the name prefix "U : : Fn : : \{f\} : : \text{Inst}\{i\} : : \text{FEIP} : : \text{ID}" plus ends with "::OutputStore". In other words, the ?? inside the curly braces matches to anything.
5.0.7 Matching variables via Wild Cards in Store-Names

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

5.0.8 Matching to Functions

$f(x_1, \ldots, x_n)$ 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 $f$ 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.

5.0.9 The Meaning of $\downarrow$

The symbol $\downarrow$ has the same semantics as $\downarrow$ 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 $\downarrow$ 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 $\uparrow$ 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 Code Time Circuits, simply choosing mutually-exclusive sets of input-pool contents, one valid input-set for each coord-element rule, will satisfy this condition.

5.0.10 The Meaning of $*$

$\mathcal{U}*$ represents the same syntax-string as $\mathcal{U}$ except that it may have been modified by some preceding 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 $\sigma$.

6 The Circuit Element Rules

6.1 Coordination Circuit Element
(note about $f$, $i$, and $n$: they are instantiated via matching the pattern under the line.)

generateUniqueSymbolInDomain $\forall J :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{FEIP} : \text{ID}\{?\text{id}\}? :: \text{LocalStore}$ (animator primitive)
generateAnotherStore $\forall U :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{FEIP} : \text{ID}\{id\} :: \text{LocalStore} \downarrow \mathcal{J}_{\text{K}}$

$\forall j \in [1..n]. (\sigma_{\text{IP}}(x_j) = \sigma_{\text{IP}}(\alpha_{\text{IP}})) \land \sigma_{\text{IP}} \in \mathcal{U}(U :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{FEIP} : \text{ID}\{?\text{id}\}? \land 
\sigma_{\text{IP}} \in \sigma_{\text{inst}}("\text{InputPoolsDraw}" + j + "\text{From}")) \land 
\alpha_{\text{IP}} \in \text{addressesIn}(\sigma_{\text{IP}}) \land 
(\sigma_{\text{IP}}.\alpha_{\text{IP}}) \notin \mathbf{A_{\text{IP}}})$ where $\mathbf{A_{\text{IP}}}$ 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

$\forall i \in [1..n]. (\alpha_i = \sigma_{\text{IP}}(x_i) \land \mathcal{J}_{\text{K}} = \alpha_i) \land \mathcal{J}_{\text{K}} = \sigma_{\text{inst}}("\text{CoordAssertion}"))$

$\sigma_1, ..., \sigma_n \models A

\langle "\text{CoordElementOf}" f(x_1, ..., x_n) "\text{instance}" i, U \rangle \downarrow \langle "\text{CoordElementOf}" f(x_1, ..., x_n) "\text{instance}" i, U * \rangle$

### 6.2 Function Circuit Element

$\mathcal{J}_{\text{K}} \in \mathcal{U} :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{FEIP} : \text{ID}\{?\text{id}\}? :: \text{LocalStore}$ (note about $\text{id}$: it instantiates to what matches in the ID field of the instantiated value of $\sigma_{\text{IP}}$)

$\mathcal{J}_{\text{inst}} K = \mathcal{U} :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{InstanceContext}$

$\forall i \in [1..n]. (\alpha_i = \sigma_{\text{IP}}(x_i) \land \mathcal{J}_{\text{K}} = \alpha_i)$ (means instantiate the $\alpha_i$ from the local store, then the $\sigma_i$ from $\alpha_i$)

$A = \sigma_{\text{inst}}("\text{CoordAssertion}")$

$\sigma_1, ..., \sigma_n \models A$

generateEmptyStore $\forall U :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{FEIP} : \text{ID}\{\text{id}\} :: \text{OutputStore} \downarrow \mathcal{J}_{\text{O}}$ $m = \sigma_{\text{inst}}("\text{numOutputs}")$

addLoc("out" + $i$) to $\sigma_O$ $\forall i \in [1..m]$ (create a location for each output... just add name, has empty value)

$\langle "\text{FuncRules}" \downarrow \sigma_{\text{inst}}("\text{FuncBody}"), [f, i, \text{id}], [\sigma_{\text{IP}}, \alpha_{\text{IP}}, \sigma_{\text{O}}], U \rangle \downarrow \langle \text{null}, [], U * \rangle$

moveStoreFromTo $\forall U :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{FEIP} : \text{ID}\{\text{id}\} :: \text{OutputStore}, U :: \text{Fn} : \{f\} :: \text{Inst}\{i\} :: \text{OP} : \text{ID}\{\text{id}\}$

deleteStore $\forall (\mathcal{J}_{\text{K}}) \downarrow$

$\langle "\text{FunctionElementOf}" f(x_1, ..., x_n) "\text{instance}" i, U \rangle \downarrow \langle "\text{FunctionElementOf}" f(x_1, ..., x_n) "\text{instance}" i, U * \rangle$

### 6.2.1 Context-Switch Rule

$\text{prim} \downarrow \{\text{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.}

\text{[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 syntaxString copied into it, remove the [ ] and remove the "RuleSet $\downarrow$ " and proceed with matching RuleSet rules.}$

 kê $\downarrow \text{resultSyntaxString, [resultStoreVars], U} * \downarrow$

A note about the temporary store: these are created so that multiple copies of the same circuit-element-syntax-string can be reduced at the same time without interfering.
6.3 Output Circuit Element

\[ \mathcal{O} \in \mathcal{U} : \{ f \} : \text{Inst} \{ i \} : \text{OP} : \{ \text{ID} \{ \text{Id} \} \} \mathcal{K} \]

(note about \( \text{Id} \): it is instantiated from matching to the ID position in the \( \mathcal{O} \mathcal{K} \) chosen)

\[ \mathcal{O}_{\text{inst}} \mathcal{K} = \mathcal{U} : \{ f \} : \text{Inst} \{ i \} : \text{InstanceContext} \mathcal{K} \]

\[ m = \sigma_{\text{inst}}(\text{"numOutputs"}) \]

\[ \forall i \in [1..m].( \mathcal{O}_{\text{inst}} = a_i \land a_i = \sigma_{\text{inst}}(\text{"out"} + i) \land a_i \neq \text{null} ) \] (\( \sigma_{IP} \), is null otherwise)

\[ \forall j \in [1..m].( \sigma_{IP}(a_j) = a_j \land \sigma_{IP} \neq \text{null} \land a_j = \sigma_{O}(\text{"out"} + j) \land a_j \neq \text{null} \land a_j = \text{generateAddrIn}(\sigma_{IP})) \]

\[ \text{deleteStore} \mathcal{U}(\mathcal{O} \mathcal{K}) \]

(\"OutputElementOf" f(x₁, ..., xₙ) \"instance" i, \( \mathcal{U} \)) \( \Downarrow \) (\"OutputElementOf" f(x₁, ..., xₙ) \"instance" i, \( \mathcal{U} \))

7 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. association 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.

7.1 The meaning of value names:

Primitive Datums:

\( a \) is an association
\( n \) is an integer
\( \text{fp} \) 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 struct container
\( \sigma \) is a container
\( \sigma^p \) is a primitive container
\( \sigma^s \) is a struct container
\( \sigma^a \) is an array container  (Q: make first position be indexed by 1 or 0?)
\( \sigma^{sys} \) is a system container
\( \sigma^{cs} \) 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
\( \mathbb{T} \) is the set of all structure-types
\( \mathbb{F}_T \) is the set of sets of fields of struct types. \( \forall t \in \mathbb{T} \) then \( F_t \in \mathbb{F}_T = \{ \forall f \in \text{fields of } t \}
\( \mathbb{O} \) is the set of sets of outputs of functions. \( \forall fn \in \mathbb{F}_n \) then \( O_{fn} \in \mathbb{O} = \{ \forall o \in \text{outputs of } fn \}
\( \mathbb{S} \) is the set of all symbols used in the program, of the form \(<\text{symbol-string}>\)
\( \mathbb{C} \) is the set of all characters
\( \mathbb{A} \) is the set of all associations, of the form \( U::\text{field-string} \)

### 7.2 General Expressions

\( e ::= \begin{align*}
& n \quad \text{for } n \in \mathbb{Z} \quad \text{which is the set of all integers} \\
& s \quad \text{for } s \in \mathbb{S} \quad \text{which is the set of all symbols} \\
& ch \quad \text{for } ch \in \mathbb{C} \quad \text{which is the set of all characters} \\
& a \quad \text{for } a \in \mathbb{A} \quad \text{which is the set of all associations} \\
& x \quad \text{for } x \in L \\
& e_1, e_2 \quad \text{for } e_1, e_2 \in \mathbb{A}\exp \\
& a.e \quad \text{for } a \in \mathbb{A}^s, e \in \mathbb{A}\exp \\
& a.fd \quad \text{for } a \in \mathbb{A}^s, T = \text{TypeOf}[U(a)], fd \in \text{fieldsOf}[T] \\
& e_1[e_2] \quad \text{for } e_1, e_2 \in \mathbb{A}\exp \\
& a[e] \quad \text{for } a \in \mathbb{A}^a, e \in \mathbb{A}\exp \\
& a[n] \quad \text{for } a \in \mathbb{A}^a, n \in [1, 2, \ldots] \\
& o \quad \text{for } o \in \text{addressesIn}[\sigma_O] \\
& e_1 \mathbb{X} e_2 \quad \text{for } e_1, e_2 \in \mathbb{A}\exp, \mathbb{X} \in \text{arity 2 arithmetic operators} \\
& e_1 \quad \text{for } e_1 \in \mathbb{A}\exp, \in \text{arity 1 arithmetic operators}
\end{align*} \)

### 7.3 Boolean Expressions
\[
\begin{align*}
b : & \equiv \begin{cases} 
\text{true} \\
\text{false} \\
\text{if } e_1 = e_2 \text{ then } e_1, e_2 \in \text{Aexp} \\
\text{if } e_1 < e_2 \text{ then } e_1, e_2 \in \text{Aexp} \\
\text{if } e_1 \leq e_2 \text{ then } e_1, e_2 \in \text{Aexp} \\
\text{if } e_1 > e_2 \text{ then } e_1, e_2 \in \text{Aexp} \\
\text{if } \neg b \text{ then } b \in \text{Bexp} \\
\text{if } b_1 \land b_2 \text{ then } b_1, b_2 \in \text{Bexp} \\
\text{if } b_1 \lor b_2 \text{ then } b_1, b_2 \in \text{Bexp}
\end{cases} 
\end{align*}
\]

7.4 Commands

\[
c : \equiv \begin{cases} 
P_{\text{ml}} = \text{new } t & \text{for } t \in \text{namesOf}[T_s], P \in [\text{null}, k, d] \\
\text{PmT=OTC} & \text{for } P \in [\text{null}, d, k], T \in [l, c], O \in [t, c], C \in [\text{null}, c] \\
\text{output } e_1 \text{ to } e_2 \text{ withTag } \{ c \} \quad & \text{for } e_1, e_2 \in \text{Aexp}, c \in \text{Comm} \\
\text{c}_1; c_2 & \text{c}_1, c_2 \in \text{Comm} \\
\text{if } b \text{ then } \{ c_1 \} \text{ else } \{ c_2 \} & \text{for } c_1, c_2 \in \text{Comm}, b \in \text{Bexp} \\
\text{a}_1, fd_1 \quad m = t \quad \text{a}_2, fd_2 \text{ and etc...} & \text{for } a_1, a_2 \in \mathcal{A}, T_1 = \text{TypeOf}[U(a_1)], T_2 = \text{TypeOf}[U(a_2)] \\
 & \text{fd}_1 \in \text{fieldsOf}[T_1], \text{fd}_2 \in \text{fieldsOf}[T_2]
\end{cases}
\]

\text{endFn}

The set of all the types, \( T_s \), 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.

8 Function Reduction Rules

These are the “FuncRules”

8.1 New rule

\[
\begin{align*}
T \in T_s \\
\langle \text{generateAnotherStore } [U(U::\text{WorkingStores}:\{?\text{Unique}\}), T], U \rangle & \Downarrow \langle a, U* \rangle \quad \text{(animator primitive)} \\
\langle \text{new } T, U \rangle & \Downarrow \langle a, U* \rangle
\end{align*}
\]

The “new” command can only appear on the RHS of an assignment command. It cannot appear in combination with any other expressions or commands.
8.2 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 \( \sigma_{ls} \), 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 \( \langle a, fd \rangle \) 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 \( \sigma_{ls} \). 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 \( \langle a, myArray \rangle \) then to \( \langle a, fd \rangle \) then the contents of the location is retrieved to give \( \langle a, fd \rangle \) and finally, the index is used to give whatever is the contents of the array position whose name is the symbol “5”.

8.2.1 Var to association
Ex: “T”
\[
fd = x \quad \text{Type}[\sigma_{ls}] = T_1 \quad T_1 \in T_s \quad fd \in \text{fieldsOf}[T_1] \quad \alpha_2 = \sigma_{ls}(fd) \quad J_{\beta, \gamma} = \alpha_2
\]
\[
\langle x, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle \alpha_2, x, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.2 Var to location
Ex: “T”
\[
fd = x \quad \text{Type}[\sigma_{ls}] = T_1 \quad T_1 \in T_s \quad fd \in \text{fieldsOf}[T_1] \quad a = J_{\beta, \gamma}
\]
\[
\langle x, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a, fd, \sigma_{ls}, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.3 Expr.Expr to Expr.Expr
\[
\langle e_1, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle e'_1, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]
\[
\langle e_2, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle e'_2, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]
\[
\langle e_1, e_2, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle e'_1, e'_2, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]

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.

8.2.4 Expr to association
\[
a = e \quad a \in A
\]
\[
\langle e, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]
8.2.5 Expr. Expr to Assoc. Expr
\[
\langle e_1, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]
\[
\langle e_1, e_2, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a, e_2, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.6 Expr. Expr to Expr. field
\[
\langle e_2, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle fd, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]
\[
\langle e_1, e_2, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle e_1, fd, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]

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.

8.2.7 Assoc. Expr to location
Type[\U(a)] = T \quad T \in T_s \quad fd \in \text{fieldsOf}[T] \]
\[
\langle a, e_1, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a, fd, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.8 Loc to association
Ex: “.myArray”
Type[\U(a_1)] = T_1 \quad T_1 \in T_s \quad fd \in \text{fieldsOf}[T_1] \quad \text{\\&}[x = \text{a}_1 \quad a_2 = \sigma_x(fd) \quad \text{\&}[y = \text{a}_2
\[
\langle a_1, fd, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a_2, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.9 Loc to value
Ex: “.myInt”
Type[\U(a_1)] = T_1 \quad T_1 \in T_s \quad fd \in \text{fieldsOf}[T_1] \quad \text{\&}[x = \text{a}_1 \quad p = \sigma_x(fd)
\[
\langle a_1, fd, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle p, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.10 expr [ expr ] to array [ expr ]
\[
\langle e_1, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \quad \text{Type}[\U(a_1)] = T_1 \quad T_1 \in T_{arrays}
\]
\[
\langle e_1, e_2, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a[e_2], \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.11 Array [ expr ] to array [ index ]
\[
\text{Type}[\U(a_1)] = T_1 \quad T_1 \in T_{arrays} \quad \langle e, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle n, \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle \quad n_2 \in \text{fieldsOf}[T_1]
\]
\[
\langle a[e], \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle a[n], \sigma_y, \sigma_{ls}, \sigma_O, \U \rangle
\]

8.2.12 Array [ index ] to value
Ex: “[ 5 ]” if type that is retrieved is *not* an association:
\[
\text{Type}[\U(a_1)] = T_1 \quad T_1 \in T_{arrays} \quad n \in \text{fieldsOf}[T_1] \quad \text{\&}[x = \text{a}_1 \quad p = \sigma_x(n)
\]
\[
\langle a_1[n], \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle \Downarrow \langle p, \sigma_x, \sigma_{ls}, \sigma_O, \U \rangle
\]
### 8.2.13 Array [ index ] to association  

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

\[
\text{Type}[U[a_1]] = T_1 \quad T_1 \in \mathcal{T}_{\text{arrays}} \quad n \in \text{fieldsOf}[T_1] \quad a_2 = \sigma_x(n_2) \quad \exists \gamma K = a_2
\]

\[\langle a_1[n], \sigma_x, \sigma_{ls}, \sigma_O, \mathcal{U} \rangle \downarrow \langle a_2, \sigma_y, \sigma_{ls}, \sigma_O, \mathcal{U} \rangle\]

### 8.2.14 Arithmetic rules (just one example to show form)

\[\langle e_1, \sigma_x, \sigma_{ls}, \sigma_O, \mathcal{U} \rangle \downarrow n_1 \quad \langle e_2, \sigma_x, \sigma_{ls}, \sigma_O, \mathcal{U} \rangle \downarrow n_2 \quad n \text{ is sum of } n_1 \text{ and } n_2\]

\[\langle e_1 + e_2, \sigma_x, \sigma_{ls}, \sigma_O, \mathcal{U} \rangle \downarrow n\]

Notice the \(\sigma_x\) 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, \(\sigma_x\) remains the same from place to place in this rule.

### 8.3 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.

### 8.4 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 LIHS has an association to a container before the assignment operation takes place. The container associated to by the LIHS 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 \(a\), copy then delete contents of each location in container assoc to by \(a\).
- 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, “\(a.fd\)” is used on both sides. One alternative would be to use an embedded notation like this:

\[
\langle (fd_1, \sigma_x, \sigma_{ls}, \sigma_O, \cup) \rangle \text{ kmn} = \text{ta} \langle fd_2, \sigma_y, \sigma_{ls}, \sigma_O, \cup \rangle \downarrow \langle \sigma_x, \sigma_{ls}, \sigma_O, \cup * \rangle
\]

which has a sigma on each side of the = sign. This is a nice notation, might use it in future.
8.5 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) \texttt{ml} = \text{temp value} \quad \text{"T.myInt ml= 5;"}

\rightarrow \text{the temporary value is placed into the location.. don’t need RHS designator}

\begin{align*}
\text{Type}[\text{null}(a_1)] &= T_1 & T_1 \in T_s & fd_1 \in \text{fieldsOf}[T_1] & \sigma_1[K] = a_1 \\
\sigma_1(fd_1) &= p \\
\{a_1,fd_1 \text{ ml} = p, \sigma_x, \sigma_{lb}, \sigma_O, \text{null}\} & \downarrow (\sigma_x, \sigma_{lb}, \sigma_O, \text{null})
\end{align*}

2. loc(a) \texttt{dml} = \text{temp assoc} \quad \text{"T.myContainer dml= new Container;"}

\rightarrow \text{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{align*}
\text{Type}[\text{null}(a_1)] &= T_1 & T_1 \in T_s,a & fd_1 \in \text{fieldsOf}[T_1] & \sigma_1[K] = a_1 \\
\text{Type}[\text{null}(a_2)] &= T_2 \\
\text{Type}[\sigma_1(fd_1)] &= a: T_{11} & T_2 &= T_{11} \\
\{\text{delete}[\text{null}(\sigma_1(fd_1))] & \mid \text{Type}[\sigma_1(fd_1)] \in a: T_s,a \} & \\
\sigma_1(fd_1) &= a_2 \\
\{a_1,fd_1 \text{ dml} = a_2, \sigma_x, \sigma_{lb}, \sigma_O, \text{null}\} & \downarrow (\sigma_x, \sigma_{lb}, \sigma_O, \text{null})
\end{align*}

3. loc(a) \texttt{kml} = \text{temp assoc} \quad \text{"T.myContainer kml= new Container;"}

\rightarrow \text{the temporary association is placed into the LHS location.. keep in existence the container associated to by previous contents of LHS.}

\begin{align*}
\text{Type}[\text{null}(a_1)] &= T_1 & T_1 \in T_s,a & fd_1 \in \text{fieldsOf}[T_1] & \sigma_1[K] = a_1 \\
\text{Type}[\text{null}(a_2)] &= T_2 \\
\text{Type}[\sigma_1(fd_1)] &= a: T_{11} & T_2 &= T_{11} \\
\sigma_1(fd_1) &= a_2 \\
\{a_1,fd_1 \text{ kml} = a_2, \sigma_x, \sigma_{lb}, \sigma_O, \text{null}\} & \downarrow (\sigma_x, \sigma_{lb}, \sigma_O, \text{null})
\end{align*}

\text{note about } \sigma_{lb}: \text{ it will be either } \sigma_1 \text{ or } \sigma_2 \text{ the alternative rule form will fix this uncertainty.}

4. assoc \{d,k\} \texttt{mc} = \text{temp assoc} \quad \text{"T.myContainer dmc= new Container;"}

\rightarrow \text{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) \texttt{ml=tlc} \text{ selected loc(v)} \quad \text{"T.myInt ml=tlc Temp.anInt;"}

\rightarrow \text{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{align*}
\text{Type}[\text{null}(a_1)] &= T_1 & T_1 \in T_s & fd_1 \in \text{fieldsOf}[T_1] & \sigma_1[K] = a_1 \\
\text{Type}[\text{null}(a_2)] &= T_2 & T_2 \in T_s & fd_2 \in \text{fieldsOf}[T_2] & \sigma_2[K] = a_2 \\
\sigma_1(fd_1) &= \sigma_2(fd_2) & \sigma_2(fd_2) &= \text{null} \\
\{a_1,fd_1 \text{ ml} = t \ a_2,fd_2, \sigma_x, \sigma_{lb}, \sigma_O, \text{null}\} & \downarrow (\sigma_x, \sigma_{lb}, \sigma_O, \text{null})
\end{align*}
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.
   \[\{a_1, fd_1, kml=ta\} \cup \{a_2, fd_2, \sigma_z, \sigma_b, \sigma_O, \emptyset\} \downarrow \langle \sigma_z, \sigma_b, \sigma_O, \emptyset \rangle\]

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.
   \[\{a_1, fd_1\} \cup \{a_2, fd_2, \sigma_z, \sigma_b, \sigma_O, \emptyset\} \downarrow \langle \sigma_z, \sigma_b, \sigma_O, \emptyset \rangle\]

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 cont-
   tainer 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.
   \[\{a_1, kmc=t\} \cup \{a_2, \sigma_z, \sigma_b, \sigma_O, \emptyset\} \downarrow \langle \sigma_z, \sigma_b, \sigma_O, \emptyset \rangle\]

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..
Type[$\mathbb{U}(a_1)$] = $T_1$  
$T_1 \in T_s$  
$\mathcal{J}_1 = a_1$

Type[$\mathbb{U}(a_2)$] = $T_2$  
$T_2 = T_1$  
$\mathcal{J}_2 = a_2$

$\forall fd_1 \in \text{fieldsOf}[T_1], \{ \text{delete}[\mathbb{U}(\sigma_1(fd_1))] \mid \text{Type}[\sigma_1(fd_1)] \in a:T_{s,a} \}$

$\forall fd_1 \in \text{fieldsOf}[T_1], \{ \sigma_1(fd_1) = \sigma_2(fd_1) \}$

$\forall fd_1 \in \text{fieldsOf}[T_1], \{ \sigma_2(fd_1) = \text{null} \}$

$\langle a_1, \text{dmc} = t, a_2, \sigma_s, \sigma_O, \mathbb{U}, \mathbb{U} \rangle \Downarrow \langle \sigma_s, \sigma_O, \mathbb{U} \rangle$

10. $\text{loc}(v)$  
$\text{ml} = \text{cl}$  
selected loc($v$)  
"T.myInt m=c Temp.anInt;"

$\rightarrow$

copy contents from location on RHS to location on LHS.

Type[$\mathbb{U}(a_1)$] = $T_1$  
$T_1 \in T_s$  
$fd_1 \in \text{fieldsOf}[T_1]$  
$\mathcal{J}_1 = a_1$

Type[$\mathbb{U}(a_2)$] = $T_2$  
$T_2 = T_1$  
$fd_2 \in \text{fieldsOf}[T_2]$  
$\mathcal{J}_2 = a_2$

$\rightarrow$

$\sigma_1(fd_1) = \sigma_2(fd_2)$

$\langle a_1, fd_1 \text{ ml} = c a_2, fd_2, \sigma_s, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_s, \sigma_O, \mathbb{U} \rangle$

11. $\text{loc(a)}$  
$\text{ml} = \text{ca}$  
selected assoc  
"T.myArray ml=ca Temp.tempArray;"

$\rightarrow$

copy association from RHS to location on LHS.

This rule is disallowed until side effects are added

Type[$\mathbb{U}(a_1)$] = $T_1$  
$T_1 \in T_s$  
$fd_1 \in \text{fieldsOf}[T_1]$  
$\mathcal{J}_1 = a_1$

Type[$\mathbb{U}(a_2)$] = $T_2$  
$T_2 = T_1$  
$T_{11} = T_2$

$\rightarrow$

$\sigma_1(fd_1) = a_2$

$\langle a_1, fd_1 \text{ ml} = c a_2, fd_2, \sigma_s, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_s, \sigma_O, \mathbb{U} \rangle$

12. $\text{loc(a)}$  
$\text{dml} = \text{ca}$  
selected assoc  
"T.myArray dml=ca Temp.tempArray;"

$\rightarrow$

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

Type[$\mathbb{U}(a_1)$] = $T_1$  
$T_1 \in T_s$  
$fd_1 \in \text{fieldsOf}[T_1]$  
$\mathcal{J}_1 = a_1$

Type[$\mathbb{U}(a_2)$] = $T_2$  
$T_2 \in T_s$  
$T_{11} = T_2$

$\rightarrow$

delete[$\mathbb{U}(\sigma_1(fd_1))] \mid \text{Type}[\sigma_1(fd_1)] \in a:T_{s,a}$

$\sigma_1(fd_1) = a_2$

$\langle a_1, fd_1 \text{ dml} = ca a_2, \sigma_s, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_s, \sigma_O, \mathbb{U} \rangle$

13. $\text{loc(a)}$  
$\text{dml} = \text{cac}$  
selected assoc  
"T.myArray dml=cac Temp.tempArray;"

$\rightarrow$

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"

Type[$\mathbb{U}(a_1)$] = $T_1$  
$T_1 \in T_s$  
$fd_1 \in \text{fieldsOf}[T_1]$  
$\mathcal{J}_1 = a_1$

Type[$\mathbb{U}(a_2)$] = $T_2$  
$T_2 \in T_s$  
$T_{11} = T_2$

$\rightarrow$

(delete[$\mathbb{U}(\sigma_1(fd_1))] \mid \text{Type}[\sigma_1(fd_1)] \in a:T_{s,a}$

$\sigma_3 = \text{copyOf}[\mathbb{U}(a_2)]$

$\sigma_1(fd_1) = a_3$

$\langle a_1, fd_1 \text{ dml} = cc a_2, \sigma_s, \sigma_O, \mathbb{U} \rangle \Downarrow \langle \sigma_s, \sigma_O, \mathbb{U} \rangle$

14. $\text{loc(a)}$  
$\text{klm} = \text{cac}$  
selected assoc  
"T.myArray ml=ca Temp.tempArray;"

$\rightarrow$

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.
Function Reduction Rules

15. assoc  \textbf{kmc=cc} selected assoc  

\[ \text{Ex: } \text{"T.myContainer kmc=cc In.aContnr;"} \]

\[ \text{Have to state the way to treat LHS.. as either a container or as a location. The \"c\" on the LHS of = means treat the LHS as a container.} \]

\[ \text{So, copy contents from RHS container to interior of LHS container..} \]

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

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

\[ \text{Ex: } \text{"T.myContainer dmc=c In.aContnr;"} \]

\[ \text{Have to state the way to treat LHS.. as either a container or as a location. The \"c\" on the LHS of = means modify the contents of LHS, which implies treat function-units and change the data each of the two sees in the third via side-effect.} \]

\[ \text{The \"c\" on the LHS of = means delete the previously-associated-to containers.} \]

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

\[ \langle b, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \text{true} \quad \langle c_1, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \langle \sigma_x, \sigma_{ls}, \sigma_O, U^* \rangle \]

\[ \langle b, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \text{false} \quad \langle c_2, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \langle \sigma_x, \sigma_{ls}, \sigma_O, U^* \rangle \]

\[ \langle \text{if } b \text{ then } \{ c_1 \} \text{ else } \{ c_2 \}, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \langle \sigma_x, \sigma_{ls}, \sigma_O, U^* \rangle \]

\[ \langle \text{if } b \text{ then } \{ c_1 \} \text{ else } \{ c_2 \}, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \langle \sigma_x, \sigma_{ls}, \sigma_O, U^* \rangle \]

\[ \langle \text{if } b \text{ then } \{ c_1 \} \text{ else } \{ c_2 \}, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \langle \sigma_x, \sigma_{ls}, \sigma_O, U^* \rangle \]

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..

8.7 Output rule

\[ o \in \text{addressesIn}[\sigma_O] \]
\[ e \downarrow a_r \quad \sigma_O(o) := a_r \downarrow \sigma_O[v := a_r] \quad \exists x, k \in \sigma \]
\[ \langle \text{"TagRules" } \ni c, \{ \sigma_x, \sigma_{ls}, \sigma_O \}, U \rangle \downarrow \langle \text{null }, [\sigma_x, \sigma_{ls}, \sigma_O, U^*] \rangle \]
\[ \langle \text{output } e \text{ to } o \text{ withTag } \{ c \}, \sigma_x, \sigma_{ls}, \sigma_O, U \rangle \downarrow \langle \sigma_x, \sigma_{ls}, \sigma_O[o := a_r], U^* \rangle \]

9 Conclusion

This paper has shown the operational semantics for the circuit-format used to distribute programs in the Code Time Parallel Programming 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.