The Big-Step Operational Semantics of the Circuit Elements of a Hardware-Independent Parallel Intermediate Format

BY

Sean Halle
seanhalle@yahoo.com

Abstract

This paper presents the operational semantics of the circuit elements of an intermediate format that has four properties. The first property is that the format does not have the concept of a persistent processor, but rather generates short-lived processors, facilitating the same program to run unchanged on one processor or many. Second, the code-length and complexity of a schedulable work-unit is automatically adjusted to be efficient on the hardware when the application is installed. Third, the data structures used in the program are divided at run-time in partnership with a hardware-specific scheduler via an application interface. The application hands a data-structure to an intermediate-format symbol that responds with how many pieces to divide the data into, each of those pieces can then repeat the process. Fourth, easy-to-use high level languages exist that compile down to the format, including a backwards compatible variant of Java.

An application expressed in the intermediate format takes the form of a recipe to build such a "dynamic" circuit. Each operator in the circuit has an optional guard and a short, atomic, non-loop-containing segment of sequential code. The operator spawns a processor to reduce a copy of its code for every set of data that makes it past the guard. Loops are created via wiring. A wire represents a causal dependency.

The processor invariant property arises from the spawning behavior of the operators. The implementation of the format is free to decide how many spawned copies of an operator overlap in time. Meanwhile, the ability to adjust complexity of schedulable units of code arises from the short atomic operators. The atomicity and the spawning behavior force all control data to be packaged with the working data as it moves from operator to operator (each operator has only local, transient, variables). This allows a boundary to be drawn around any arbitrary segment of the circuit and the inside mapped onto a sequence of operations (with inserted tests), suitable for execution on a sequential processor.

The semantics of the circuit elements are presented in a modified form of Big Step semantics. Companion papers state the semantics for building the circuit, and give the semantics of the sequential language used to state the behavior of operators.

1 Introduction

The difficulties historically associated with parallel programming are becomming increasingly urgent to solve. A practical solution requires achieving a satisfactory level in the goals of "write once, run high-performance anywhere", high programmer productivity, and wide-use (meaning an accepted, uniformly implemented, standard).

High performance has required the application programmer to have knowledge of the hardware and make choices of how to write the application based on that knowledge. This both reduces programmer productivity and ties the application to a certain range of hardware that it runs well on. It reduces productivity, in part, because the hardware is often quite complex, creating a steep and long learning curve for application programmers. In practice, programmers either lose productivity while learning the hardware and while tweaking the code for that hardware, or else they, more often, only gain a minimal working knowledge of it and don't produce especially efficient code. Either way, source-level modifications are necessary to update the software for new hardware, or to run the software on other machines, both of which are costly and slow.

Because of these problems, and more, a solution is desired whereby a program can be written once and have it run high performance on a number of different machines from the same distributed installation-bundle.

Many attempts to achieve this have been tried [18] [10] [13] [4] [3] [11] [2] [12] [17] [8] [14] [15] [7] [1] [6] [16] [5]. However, better solutions are still desired.

This paper gives the formal semantics of an intermediate format that can be used as one component in a software platform that focuses on the write-once run high-performance anywhere goal.

The syntax of the intermediate format describes how to build a circuit. The constructed circuit performs the behavior stated by the original program. The circuit spawns short-lived processors that each apply one function to one set of data. Each function is straight-line imperative code (no loops), so a spawned processor executes a short, atomic, segment of sequential code.

The Big Step operational semantics are extended, then used to state the semantics of the circuit elements constructed by the proposed intermediate format, part II gives the semantics of the straight-line code executed by the function-processors spawned by the circuit.

Section 2 describes the elements of a circuit and how they work. Section 3 explains the extensions to the Big Step notation. Section 4 gives the formal semantics of the circuit elements in that notation. Section 5 discusses related work, and section 6 concludes.

2 The Circuit Elements

A circuit is composed of units connected by wires. Two kinds of unit are proposed: function-units and processor-units. Function-units are composed of three circuit elements, and spawn short-lived function-processors. Processor-units are black boxes used to communicate with processors outside the circuit's name-space. The circuit has its own name-space. Each unit has a name in the circuit's name-space, and each input is a sub-name of the unit. Wires invoke the names of inputs of units. The placement of a wire tells the output of the up-stream unit the name of the input of the down-stream unit. The up-stream unit uses the circuit's name-space mechanism to place each output-datum into the named input of the down-stream unit.

Terminology In the terminology[9] we adopt for this paper, a running program is considered a processor. The program itself is considered a specification of a processor. The run command uses the specification to create a *derived* processor that behaves according to the specification.

A created (derived) processor, call it processor A, can have parts of its behavior performed by other processors. The processors reachable from processor A's name-space are said to be in the name-space of processor A. Any of them may be called upon to perform a sub-problem for processor A. In addition, the other processors can communicate with each other by using processor A's name-space mechanism. For example, a network is a name-space mechanism, as is a bus inside a silicon-chip, and so can a physical memory be used as a name-space mechanism.

Processor B, is said to be contained within processor A if B is in the name-space of A. B can then use A's name-space mechanism to communicate with any other processors in A's name-space.

In order for processor A to communicate with a processor that is outside its own name-space, it must use a containing processor's name-space. For example, in order for one running program to communicate with another, it must use the running OS instance's name-space mechanism, such as an inter-process communication call. Both running programs are processors in the running OS instance's name-space, and the inter-process call is the mechanism that implements the name-space's behavior.

Processor Units A processor-unit is used in a circuit the way an inter-process call is used. It allows a running program to communicate with any other processor contained in the OS instance's name-space. A processor-unit's behavior is implemented for specific hardware as part of implementing the intermediate format for that hardware.

To use a processor-unit to communicate, first give it the name of the target processor. This causes the target processor to be "connected" to the processor-unit symbol. Then, hand commands to the processor-unit, and out from the processor-unit come responses from the target processor.

All OS services are gained through processor-units. An application uses the name of an OS-supplied processor to connect to that processor. The application then sends commands to the OS-service-providing-processor and receives responses via the processor-unit symbol.

Function Units A function-unit's behavior is defined by the programmer. It contains three primitive circuit elements: a coordination-element, a function-generator-element, and an output-element.

The coordination-element deals with data coming in to the function-unit. The function-generator-element creates short-lived processors, each of which is handed a set of inputs from the coordination-element. Each created processor executes the function-code on one set of inputs, and thereby creates outputs. When the segment of code is done, the outputs are handed to the output-element and the processor disappears. The output element then distributes the outputs to the coordination-elements of connected down-stream units.

2.1 Details of Units

2.1.1 Function Unit Details.

The coordination-element contains one or more pools of input-data, one pool for each wire coming in to the function-unit. When data enters, it comes off a wire and enters the corresponding input-pool.

The coordination-element includes a guard boolean. It inspects the data in its various input-pools, looking for any combination that satisfies the boolean. The boolean was written by the programmer. Its terms are locations in the data-structures taken from the input-pools. The coordination-element consumes the input data (removes it from the input pools), generating sets of input-datums, such that every set satisfies the boolean.

Each input-set is put in the function-generator-element's pool of input-sets. The function-generator element removes input-sets and creates a function-processor for each. A function-processor is created containing a copy of the code the programmer wrote for the function-unit. It performs the code-segment on the input-set, constructing a set of output data in the process. When the code is done (reduced to null), the output-set is placed in the output-elements' pool of output-sets and the function-processor disappears.

The output-element consumes the sets from its pool. Each element of the set is a down-stream-input-name and value pair. The output-element communicates the output values by placing each value into the input-pool of the named input of the down-stream unit.

Ordering The overlap of function-processor existence is not defined. Existing one after the other is just as valid as millions existing together, all performing overlapped computation. Likewise the order of input-sets being taken out of input-pools is not defined, nor are any a-priori constraints on which datums may be grouped together. The only such constraints are the ones the programmer specifies via the boolean in the coordination-element.

The only ordering defined is causality. Datums arrive in input-pools before they appear as elements of input-sets. A function-processor is created after its input-set appears in the function-generator-element's input-set-pool. Datums in output-sets arrive in the output-set pool of an output-element before they arrive in input-pools of down-stream coordination-elements.

2.1.2 Processor-Unit Details.

Processor-units are black-boxes. They only have input-pools defined, one input-pool for each incoming wire. An up-stream unit places datums into the input-pool named by the shared wire.

Up-stream units have no means to tell how many datums are in down-stream input-pools, for any units, including processor-units.

Output datums that come from processor-units simply appear in the input-pools of down-stream units. No timing or ordering exists other than causality. A response datum will not appear in a down-stream input pool before the corresponding command-datum is placed in an input-pool of the processor-unit.

2.2 Memory model

The memory is split into multiple independent name-spaces, called containers. Each container is analogous to a separate virtual-memory space, and is a separate memory-processor. A container is denoted with the standard symbol for a store: σ.

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

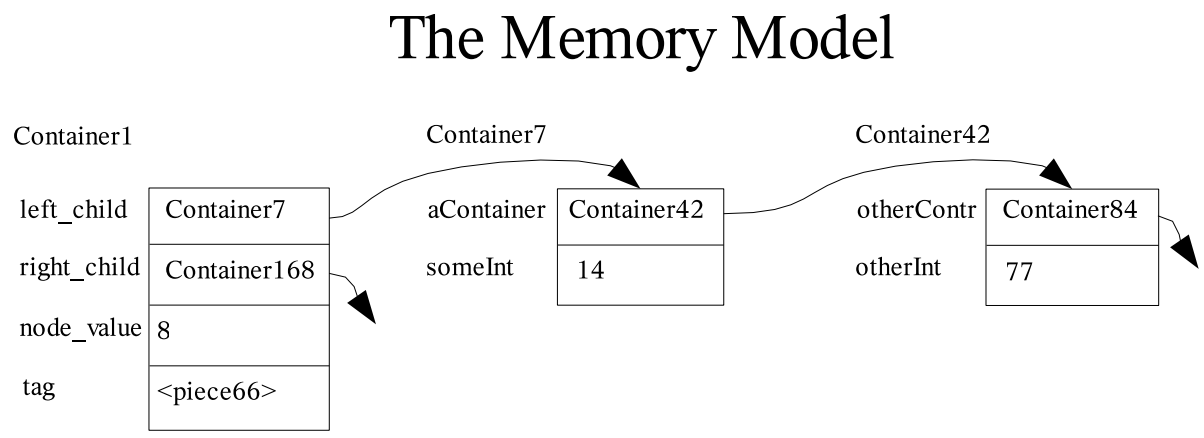

Figure 1. Each box represents a container, which is a separate address space (separate memory-processor). The labels above the boxes are the names of the address spaces (processors). The labels to the left of each box are the addresses. To the right of each address, inside the box is the memory contents.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

2.3 Computation Sequence

The basic sequence of computation is:

→ Containers land in the input pools inside a coordination element

→ The coordination-element performs, as one atomic operation:
- chooses a set of containers from the input pools that satisfies the boolean
- creates an empty Function Execution Instance (FEI)
- creates an empty local-store, with parameter locs and temp vars, in the new FEI
- places the chosen input containers' names in the local-store's parameter locations
- places the FEI in the function-generator-element's FEI Pool

→ The function-generator-element performs, as one atomic operation:
- chooses an FEI from the FEI pool
- places a copy of the function syntax-string into that FEI
- creates a processor to reduce the function syntax-string, and starts it

→ The created processor does, as one atomic operation:
- reduces the function syntax-string
- during reduction, modifies the temporary variable locations in the local store
- during reduction, may modify the containers reachable from the input-set
- during reduction, adds output-pairs to the output locations in the FEI
- when reduction is complete, places the FEI in the output-element's pool
- the created processor then disappears

→ The output-element performs, as one atomic operation:
- takes an FEI from its pool
- takes each pair from the output locations
- looks at the name of the pair's destination input-pool
- places the pair's value into that input pool
- when all output-pairs have been removed, deletes the FEI and all its stores

3 Modifications to the Big Step Semantics

3.1 Why New Notation?

The proposed intermediate format uses concepts that are not representable with the standard big step notation.

In particular, notation is needed for the following concepts:

- name-spaces
- name-spaces that contain other name-spaces
- a root name-space (root for the circuit)

- processors communicating across name-spaces
- the concept of *derived* processor, defined as a collection of: a name-space, a specification, a deriving processor, and a local state. The deriving processor derives the behavior of the specification, which causes the local state to be modified and communication events to be generated. Local state includes bookeeping information used by the deriving processor, a lookup table, received data, and a working-pattern-holder.
- Distinction between stores and name-spaces
- A means of locating name-spaces and stores starting from the circuit's root name-space
- A compound name that indicates a traversal of name-spaces
- A store's name as a compound name that starts with the circuit's root name-space
- A means of matching variables to segments of a compound name
- A represention of a circuit-element, which does not reduce, that fits into the framework of reduction rules and syntax-directed semantics
- A notation to distinguish between a store itself, the compound name of a store, a rule-variable that instantiates to a store, and a rule-variable that instantiates to the compound name of a store
- A notation to indicate whether a rule operates on the compound name of a denoted store, operates on the store entity, or operates on the contents of the store
- A means to instantiate one rule-variable to a portion of the rule-stated name of another variable
- Reduction rules that have an imperative nature. The antecedents have side-effects, and are evaluated, within the rule-statement, top-down left-to-right (this implies that the derived processor must have bookeeping state that the deriving processor uses during reduction of a single rule)
- A notation for potential modification via side-effect, from one part of a rule to another
- A notation for the creation of a derived processor

3.1.1 Variable substitution and instantiation

Multiple levels of instantiation may take place within a single rule. For example, in "$\sigma_{\mathrm{IP}j}$", two levels of instantiation are taking place. First the "j" is replaced inside the syntax-string of the variable *name* then, that name is instantiated to a value. For example, the name "σ_{IP1}" instantiates to a value. This is the same value that "$\sigma_{\mathrm{IP}j}$" instantiates to when j has been instantiated to 1. First, j instantiates to 1, then $\sigma_{\mathrm{IP}j}$ instantiates to σ_{IP1}, then σ_{IP1} instantiates to some value.

3.1.2 The Root Name Space, R, and the set of all processors reachable from root, $\mathcal{R}$

A name-space is part of a processor. A circuit, as a whole, is a processor, and so it has a name-space, called the root name-space and is denoted as R. The set of all processors reachable from the circuit's root name-space is denoted $\mathcal{R}$.

A store is a processor that has write and read commands. So, all stores used in the circuit's behavior are reachable from R, and are included in the set $\mathcal{R}$.

Multiple types of store exist in the circuit:

- those that hold program-syntax while it is being reduced
- those that are created and destroyed explicitly by program constructs (containers)

- local stores that hold the parameters and temporary variables of a generated-function
- stores inside a derived processor, that the deriving processor uses to communicate values from one part of a rule to another
- and so on

3.1.3 Compound names

Each store has a name that its containing name-space uses to communicate with it. That name-space is in turn inside another name-space which is inside another and so on, up to the circuit's root name-space. The sequence of names of name-spaces is called a compound-name. Here's an example of a compound name: $R::\mathrm{Fn}::\mathrm{myFunc}::\mathrm{Inst8}::\mathrm{FEIP}::\mathrm{ID3}::\mathrm{OutputStore}$.

The fields of a compound-name are separated by :: making a directed graph structure, where each field specifies an edge in the graph. $R::$ represents the root name-space, and begins the compound-name of every processor (store) in the circuit. Each field following the $R::$ is the local name of a name-space. So, "Fn" is a name recognised by the root name-space. The thing communicated with by using the name "R::Fn" is the name-space that contains the names of all the function-units in the circuit. If the intermediate format has a function-unit named "myFunc1" in it, then "myFunc1" will be a name recognised by the "R::Fn" name-space. Likewise, specifying "$R::\mathrm{Fn}::\mathrm{myFunc1}$" communicates with the name-space that knows all the instances of that function, and so on.

As an example, $R::\mathrm{Fn}::\mathrm{myFunc1}::\mathrm{Inst4}::\mathrm{FEIP}::\mathrm{ID23}::\mathrm{OutputStore}$ is the compound-name that communicates with the output store of the 23rd generated function of the 4th instance-in-the-circuit of the myFunc1 function.

A name-space is inside a processor, so technically, saying that a compound-name communicates with a name-space is not correct. The communication is actually with the processor that the name-space is a part of. We usually say the name-space is communicated with when the processor that name-space is part of has no meaniful commands, it's only use is communication via the name-space.

For the curious, the circuit's root name-space is in turn inside the OS instance's name-space. The OS instance is a processor in and of itself, and exists inside multiple levels of name-space that are the local area network, wide-area network and the internet, if it is connected. If it is not connected, then its physical presence acts as a default name (in the "Universal" physical-name-space).

3.1.4 Stores

σ represents a store, which is a type of processor that only has write and read commands. In syntax-driven semantics, a store can be considered a syntax-string of name-value pairs plus a set of store-processor primitives for generating unique new names, generating new pairs, deleting pairs, changing the value of a pair whose name matches a given name, and returning the value of a pair whose name matches a given name. Giving a name, or value, or other command to a store is communicating to it. Receiving a returned value is communication from the store.

3.1.5 Black board braces: operating on the compound-name vs the processor named

Black board braces appearing in a rule indicate that a compound-name itself is being operated on by the rule, rather than the processor (name-space, store) named. For example $R::\mathrm{Fn}::\mathrm{myFunc1}::\mathrm{Inst4}::\mathrm{FEIP}::\mathrm{ID23}::\mathrm{OutputStore}$ is the compound-name of the output store of the 23rd generated function of the 4th instance-in-the-circuit of the myFunc1 function. When this name appears in a rule, it means "communicate with the thing named by this compound-name". However, sometimes one wishes to write a rule that operates on the name-itself. In such cases, one places the name inside black board braces like this:

$[\![R::\text{Fn}::\text{myFunc1}::\text{Inst4}::\text{FEIP}::\text{ID23}::\text{OutputStore}]\!]$

Anything placed inside black board braces has the meaning that the compound-name itself is operated on. For example, if "σ_o" appears in a rule, it means the contents of the denoted store is what the rule operates on. However, if it appears as "$[\![\sigma_o]\!]$" then the compound-name used to communicate with that store is what the rule operates on.

So, if one desires to explicitly affect which store a σ in a rule instantiates to, or retrieve the compound-name to that store, σ is placed inside black-board brackets, like this:

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

To instantiate σ to a particular compound-name, do this:

$[\![\sigma_o]\!]=[\![R::\text{Fn}::\text{myFunc1}::\text{inst4}::\text{FEIP}::\text{ID23}::\text{OutputStore}]\!]$

3.1.6 Rule-variables that mean a compound-name rather than the processor named

Sometimes, one desires a variable in a rule to be instantiated to a compound-name or part of a larger compound-name. This form of variable always means "operate on the name itself not what the name is used to communicate with". Such variables appear in black board type, like this:

$\mathbb{a}$

During reduction of a rule, such a variable is replaced by its instantiated compound-name string in the syntax-string being reduced.

Because σ represents communication with a store, the form "σ" cannot be used to instantiate the compound-name of the store to anything in the syntax-string being reduced. That is the purpose of $\mathbb{a}$.

To instantiate an $\mathbb{a}$ to the compound-name that retrieves σ, do this:

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

When appearing in a rule, this expression could have other meanings as well: 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.

The syntax $\mathcal{R}(\mathbb{a})$ means communicate with the processor denoted by the compound-name. So, "σ" appearing in a rule is the same as "$\mathcal{R}(\mathbb{a})$" appearing in a rule when $\mathbb{a}=[\![\sigma]\!]$ is true.

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

σ or $\mathcal{R}([\![\sigma]\!])$ or $\mathcal{R}(\mathbb{a})$

3.1.7 Wild cards in compound-name strings

$[\![R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{??\}::\text{OutputStore}]\!]$ means all compound-names 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 remaining terms are literals and match exactly. Thus, this expression means all compound-names of output stores, that have the compound-name prifix "$R::\text{Fn}::$" followed by whatever $\mathbb{f}$ instantiated to, follwed by "Inst" then whatever $\mathbb{i}$ instantiated to then ":: FEIP: : ID" then any integer, then "::OutputStore". The ?? inside the curly braces match to any integer.

3.1.8 Literal strings

The term ″numOutputs″ appearing in a rule is a literal string. What's inside the quotes is the set of symbols that the rule matches to verbatim ("match" happens in whatever symbol-embodiment is used by the deriving processor that is reducing the rule).

3.1.9 Instantiating rule-variables via Wild Cards

As an example of use, in "$\sigma_O \in \mathcal{R}(R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\mathbb{id}?\}::\text{OutputStore})$" the $\mathcal{R}(R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\mathbb{id}?\}::\text{OutputStore})$ resolves to the set of all stores whose names match the pattern. From this set, one store is chosen and $[\![\sigma_O]\!]$ is instantiated to its name. At the same time, $\mathbb{id}$ is instantiated to what comes between "$R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}$" and "$::\text{OutputStore}$" in the name instantiated to $[\![\sigma_O]\!]$.

3.1.10 Matching to function names

$\mathbb{f}(x_1, ..., x_n)$ 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 into $\mathbb{f}$, each i subscript in x_i instantiates to a number, and each resulting x_1, x_2 and so on instatiate to a parameter-name, and n instantiates into the number of parameters. Thus, a function with any number of parameters will match to this form.

3.1.11 The Meaning of ⇳

The symbol ⇳ has the same semantics as ⇓ except that the RHS must always be an exact copy of the left-hand side, 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 set of stores affected in one atomic update.

The ordering of datums being consumed from input pools cannot be predicted. However, it is a definite order. Whatever the order that actually occurs, all antecedents hold with that order.

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, simultaneously, many copies of the same ⇳ rule can post. Their antecedents will all remain valid.

The requirement that antecedents are valid in the order of actual posting is satisfied by choosing mutually-exclusive sets of data that each rule-reduction "consumes". In other words, a rule-reduction removes a set of datums from the pools in a coordination circuit element.

3.1.12 The Meaning of ∗

$\mathcal{R}*$ means "zero or more modifications to $\mathcal{R}$". $\mathcal{R}*$ represents the same set of processors, each with the same state, as just $\mathcal{R}$ except that one or more processors, usually stores, may have modifications. The $*$ version always appears on the right-hand side of a rule, with a corresponding un-starred version on the left. This arrangment has the meaning that some portion of the rule might or might not have modified state of a processor in $\mathcal{R}$. The antecedents of the rule will indicate where the modification may have been performed. A $*$ indicates that an effect of the rule, if it has happened, has happened by side-effect. Another way of viewing side-effects is as a communication event between separate processors. Thus, the $*$ is indicating that the rule's effects happen via communication between processors. The $*$ designator may also be applied to a σ (each store is a separate memory-processor).

4 The Circuit Element Rules

The new primitives introduced:

To get a name that is unique from all the others in the circuit, but matches a pattern:
generateUniqueNameMatching $[\![R::\text{Fn}::\text{myFunc1}::\text{Inst4}::\text{FEIP}::\text{ID}\{?\text{Unique?}\}]\!]$

To bring a new memory processor (store) into existence:
generateTheStore $\mathcal{R}(R::\text{Fn}::\text{myFunc1}::\text{Inst4}::\text{FEIP}::\text{ID23}::\text{LocalStore})$

To change the compound-name of a given processor (ie, move it):
moveStoreFromTo $(R::\text{Fn}::\text{myFunc1}::\text{Inst4}::\text{FEIP}::\text{ID23}::\text{OutputStore},\ R::\text{Fn}::\text{myFunc1}::\text{Inst4}::\text{OP}::\text{ID23})$
which moves an output store from the Function-Execution-Instance pool of the 4th instance of myFunc1 to the OuputPool that lives inside the Output Element of that instance of that function.

To destroy a store that σ instantiates to, do this:
deleteStore $\mathcal{R}([\![\sigma]\!])$ or deleteStore σ or deleteStore $\mathcal{R}(\mathbb{a})$

To add a name-value pair to the output-store, that has the given name but empty value:
addPairNamed($''$out2$''$) to σ_O

To create a new derived processor:
$\looparrowright \text{syntax}-\text{string}-\text{of}-\text{func}, \text{myFunc1}, 4, 23, \sigma_x, \sigma_{\text{ls}}, \sigma_O, \mathcal{R}$
This creates a new derived processor that exhibits the behavior of whatever "syntax-string-of-func" is, and gives it a compound name with myFunc1 for function-name, 4 for instance-number, and 23 for ID number.

4.1 Coordination Circuit Element

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

generateUniqueNameMatching $[\![R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\text{Unique?}\}]\!] \Downarrow \mathbb{id}$ (primitive)
generateTheStore $\mathcal{R}(R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{\mathbb{id}\}::\text{LocalStore}) \Downarrow [\![\sigma_{\text{ls}}]\!]$ (primitive)

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

$\forall i \in [1...n].(\ \mathbb{a}_i = \sigma_{\text{ls}}(x_i)\ \wedge\ [\![\sigma_i]\!] = \mathbb{a}_i\ )$ (the σ_i are the containers making up the input-set)
$A = \sigma_{\text{inst}}(''\text{CoordAssertion}'')$ (A is the "guard" boolean)
$\sigma_1, ..., \sigma_n \vDash A$ (the data reachable from the input set must satisfy the guard boolean)

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

4.2 Function-Generator Circuit Element

$[\![\sigma_{\text{ls}}]\!] \in [\![R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{?\mathbb{id}?\}::\text{LocalStore}]\!]$ (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)
$[\![\sigma_{\text{inst}}]\!] = [\![R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{InstanceContext}]\!]$
$\forall i \in [1...n].(\mathbb{a}_i = \sigma_{\text{ls}}(x_i) \ \wedge \ [\![\sigma_i]\!] = \mathbb{a}_i)$ (the σ_i are the containers making up the input-set)
$A = \sigma_{\text{inst}}(''\text{CoordAssertion}'')$ (A is the "guard" boolean)
$\sigma_1, ..., \sigma_n \vDash A$ (the data reachable from the input set must satisfy the guard boolean)

generateTheStore $\mathcal{R}(R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{FEIP}::\text{ID}\{\mathbb{id}\}::\text{OutputStore}) \ \Downarrow \ [\![\sigma_O]\!]$
$m = \sigma_{\text{inst}}(''\text{numOutputs}'')$
addPairNamed($''\text{out}'' + i$) to σ_O $|\forall i \in [1...m]$ (create a name-value pair with name, but null value)

$\langle \looparrowright \sigma_{\text{inst}}(''\text{FuncBody}''), \mathbb{f}, \mathbb{i}, \mathbb{id}, \sigma_{\text{ls}}, \sigma_O, \mathcal{R} \rangle \ \Downarrow \ \langle \mathcal{R} * \rangle$

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

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

4.2.1 Create Derived Processor Rule

prim $\looparrowright$:{Make derived processor's state, which is a temporary store (different from σ_{ls})
Place syntaxString, $\mathbb{f}, \mathbb{i}, \mathbb{id}$ into the temporary store.
Pair to the store a deriving processor that accepts the interface of the syntaxString's terms-of spec.
Cause reduction of syntaxString to start.
Ouput operations in the syntaxString place name-value pairs into σ_O, (which communicates them).
Reductions continue until reach null string (else exception).

When reach null string:
delete temporary store
de-assign deriving processor (if it persists, then it's freed, else it dies)
}

$\langle \looparrowright \text{syntaxString}, \mathbb{f}, \mathbb{i}, \mathbb{id}, \sigma_{\text{ls}}, \sigma_O, \mathcal{R} \rangle \ \Downarrow \ \langle \mathcal{R} * \rangle$

The temporary store is the state of the derived processor. Because a new derived processor is created for each input-set, multiple copies of the same function-syntax-string can be reduced simultaneously without interference.

4.3 Output Circuit Element

$[\![\sigma_O]\!] \in [\![R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{OP}::\text{ID}\{?\mathbb{id}?\}]\!]$
($\mathbb{id}$ instantiates to the value in the ID position of the name of the σ_O chosen)
$[\![\sigma_{\text{inst}}]\!] = [\![R::\text{Fn}::\{\mathbb{f}\}::\text{Inst}\{\mathbb{i}\}::\text{InstanceContext}]\!]$
$m = \sigma_{\text{inst}}(''\text{numOutputs}'')$
$\forall i \in [1...m].(\, [\![\sigma_{\text{IPi}}]\!] = \mathbb{a}_i \mid \mathbb{a}_i = \sigma_{\text{inst}}(''\text{out}'' + i) \ \wedge \ \mathbb{a}_i \neq \text{null} \,)$ ($\sigma_{\text{IP}i}$is null otherwise)
$\forall j \in [1...m].(\, \sigma_{\text{IP}j}(a_j) = \mathbb{a}_j \mid \sigma_{\text{IP}j} \neq \text{null} \ \wedge \ \mathbb{a}_j = \sigma_O(''\text{out}'' + j) \ \wedge \ \mathbb{a}_j \neq \text{null} \ \wedge \ a_j = \text{generateAddrIn}(\sigma_{\text{IP}j}) \,)$
deleteStore $\mathcal{R}([\![\sigma_O]\!])$

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

5 Related Work

Large Grain Data Flow has a similar structure. However, the proposed intermediate format generalizes the semantics of data flow by introducing guards, gains benefits from only allowing straight-line segments of code inside an operator, has the notion of independent un-ordered sets on wires rather than synchronized ordered sets across wires, limits operation-code to only local variables, includes bookeeping information to define independent tasks within the data passed between operations, has the application-to-run-time-scheduler direct interface, and spawns processors to perform the behavior of the operator. These extensions support the three goals, whereas classical Data-Flow and Large Grain Data Flow lack features or have features that block attaining the goals.

The Java Virtual Machine defines a byte-code format, gcc uses multiple internal intermediate formats, and abstract machines such as the lambda calculus and PRAM exist. However, those intermediate formats have several short-comings. Their semantics imply a number of processors. The JVM, for example, implies a single processor (thread), with instructions that explicitly create additional processors (threads). The lambda calculus also implies the creation of short-lived processors (each lambda reduction can be viewed as a one-use processor), and the call-by-name semantics allow some overlap of the existence of these. 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 of controlling interactions between processors. For example, MPI exposes explicitly the existence of processors between which messages are sent. CSP and pi-calculus likewise define processors (sequential processes) that communicate. Each process is a lambda-calculus abstract machine with well-defined ordering between communication events and is a single processor. Thus, these other intermediate formats are not invariant to the number of physical processors. Invariance, here, means that the number of physical processors the code runs on is choosable independently of the code contents.

Source languages are important for programmer productivity. It is important to have a familiar, natural, high-level language that compiles to the intermediate format. Imperative, object oriented and ML like languages have all been designed for compilation to the proposed intermediate format. A single, simple, extension is needed for Java-like polymorphic behavior, and a second for higher-order functions with a polymorphic type system such as in OCAML. The languages can be designed to expose the circuit nature of the intermediate format to varying degrees. More exposure of the circuit-nature provides more opportunity for efficiency, but the essential flavor of the language stays intact. For example, CTJava is a backwards-compatible version of Java that compiles down to the proposed intermediate format.

6 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.

Bibliography

[1] G. Berry and G. Boudol. *The chemical abstract machine*. ACM Press, 1989.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[3] B. Chamberlain, S. Choi, E. Lewis, C. Lin, L. Snyder, and W. Weathersby. Zpl's wysiwyg performance model. *hips*, 00:50, 1998.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538. ACM Press, 2005.

[5] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 164–175. ACM Press, 1994.

[6] S. Fortune and J. Wyllie. Parallelism in random access machines. *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, 1978.

[7] M. P. I. Forum. *MPI: A Message-Passing Interface Standard*. 1994.

[8] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[9] Sean Halle. A mental framework for use in creating hardware-independent parallel languages. `http://codetime.sourceforge.net/content/CodeTiime_Theoretical_Framework.pdf`.

[10] Wilhelm Hasselbring. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv.*, 32(1):43–79, 2000.

[11] Paul Hilfinger and et. al. The titanium project home page. `http://www.cs.berkeley.edu/projects/titanium`.

[12] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[13] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[14] C. H. Koelbel, D. Loveman, R. Schreiber, and G. Steele Jr. *High Performance Fortran Handbook*. MIT Press, 1993.

[15] J McGraw, S. Skedzielewski, S. Allan, and R Odefoeft. *SISAL: Streams and Iteration in a Single-Assignment Language: Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, 1985. Manual M-146 Rev. 1.

[16] R. Milner. *A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[17] A. P. Reeves. Parallel pascal – an extended pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1:64–80, aug 1984.

[18] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.