A GRAINLESS SEMANTICS FOR THE HARPO/L LANGUAGE

Theodore S. Norvell

Memorial University of Newfoundland
Computer Engineering Research Labs
St. John's, NL
theo@mun.ca

ABSTRACT

This paper presents a dynamic semantics for the parallel language HARPO/L, based on Reynolds's grainless approach [1]. It shows that the approach scales to somewhat more sophisticated languages with few changes, while providing a solid semantics for the language, which will be used as a basis for compilation and optimization.

Index Terms— parallel languages, concurrency, language semantics, grainless semantics

1. INTRODUCTION

HARPO/L is a new behavioural language intended for programming configurable hardware and microprocessors [2]. The language supports interprocess communication via shared variables and rendezvous.

The meaning of shared variable programs is conventionally dependent on the granularity of operations. Consider

$$(\mathbf{co}\ \mathsf{x} := \mathsf{x} + 1 \ || \ \mathsf{x} := \mathsf{x} + 1\ \mathbf{co}) \qquad (1)$$

where x is an integer variable. If we consider that each assignment command is atomic, then (1) means $\mathsf{x} := \mathsf{x} + 2$; if we consider that each integer access is atomic, then the command nondeterministically increments x by either 1 or 2; if we consider that each memory byte access is atomic, the meaning might depend on the detailed representation of integers.

Any granularity assumption puts severe constraints on optimization. Arrays and dynamically assigned pointers allow locations to have numerous aliases, and this can inhibit almost all optimizations that involve reordering memory accesses. Worse, any granularity assumption above that provided by the native hardware requires locks to be inserted that can largely kill the benefits of parallelism. If each assignment is atomic, then a command $\mathsf{a}[\mathsf{i}] := \mathsf{a}[\mathsf{i}] + 1$ may have to be compiled as $\mathrm{lock}(s); \mathsf{a}[\mathsf{i}] := \mathsf{a}[\mathsf{i}] + 1; \mathrm{unlock}(s)$, where s is some compiler generated lock.

For these reasons, we adopt the grainless approach of [1]. This approach defines all programs that contain data races to be "wrong". For example

$$(\mathbf{co}\ \mathsf{a}[\mathsf{i}] := \mathsf{a}[\mathsf{i}] + 1 \ || \ \mathsf{a}[\mathsf{j}] := \mathsf{a}[\mathsf{j}] + 1\ \mathbf{co})$$

is "wrong" for initial states where $\mathsf{i} = \mathsf{j}$. This puts the onus on the programmer to ensure the absence of data races in their programs. One way the programmer can do so is by using explicit locks, replacing races for data with races for locks.

The rest of this paper outlines an approach to formalizing the dynamic semantics of HARPO/L. A complete semantics of the language is being developed as part of a complete description of the language. In adapting the semantic approach of [1], the following differences arise: First, HARPO/L is a more complex language than the toy language considered in [1], including, for example, objects, complex lvalue expressions, expressions that can go wrong, and rendezvous. Second, HARPO/L allows read/read data races. Third, I take a different approach to infinite computations and to computations that go wrong; these differences will be discussed further in Section 5.

We proceed as follows: In Section 2 we discuss the context in which programs are executed. This includes the object structure of the program and the data states of the program. Section 3 deals with lvalue expressions and value expressions. The semantics here is almost a traditional denotational semantics, but includes a twist to account for the fact that we need to track which locations are needed to evaluate each expression. In Section 4, we map commands to trace sets, which are sets of sequences of actions, each action being atomic. Following [1], I represent operations that access memory as two atomic actions, one indicating the start of the operation and one representing the end of the operation. I also give a semantics to the traces. Finally, Section 5 compares our work with related work and points the way to future developments.

2. CONTEXT

2.1. Object graph

HARPO/L is a static, object-based language, meaning that, while programs are structured in terms of classes, all object

The author would like to thank NSERC for funding this work.

instantiation is done statically (at compile-time). The fully semantics of the language involves three levels: the first level explains the specialization of generic classes to specific classes; the second level explains the instantiation of classes as objects; the third level explains the dynamic semantics of objects. This paper deals only with the dynamic semantics of the language and so, as a starting point, we consider a data structure that might be the output of the front-end of a compiler. This data structure consists of an object graph, a single command, and an initial state. The purpose of the dynamic semantics is to give the meaning of this command in the context of the object graph.

An object graph consists of a number of sets and functions. N is a set of identifiers. L is a set of locations. O is a set of objects. A is a set of arrays. We will also use a set V, which is the set of primitive values for a particular implementation. The special value $\bigstar$ indicates an expression computation "gone wrong" and we'll assume that it is outside any of the above sets.

Each object o has a fixed set of field names $\mathrm{fields}(o) \subseteq_{\mathrm{fin}} N$ and for $n \in \mathrm{fields}(o)$, $\mathrm{field}(o,n) \in O \cup L \cup A$. We'll extend the field function to wrong values by defining $\mathrm{field}(\bigstar, n) = \bigstar$, where $n \in N$. We assume that all local variables have been replaced by object fields, so local variables are also accessed as fields.

Similarly, for each $a \in A$, there is a length $\mathrm{length}(a) \in \mathbb{N}$, and for $v \in \{0, ..\, \mathrm{length}(a)\}$,[1] $\mathrm{index}(a, v) \in O \cup L \cup A$. For all other $v \in V$, define $\mathrm{index}(a, v) = \bigstar$. We'll also define $\mathrm{index}(\bigstar, v) = \mathrm{index}(a, \bigstar) = \bigstar$.

The set of global objects is represented by a finite partial function $\mathrm{global} \in N \rightsquigarrow O \cup L \cup A$.

Because of the static nature of HARPO/L, the global, fields, field, methods, length, and index functions do not depend on the state.

2.2. States

A state is a partial function $\sigma \in \Sigma = (L \rightsquigarrow V)$. I'll consider each state to be equal to its graph, i.e. to the set of pairs that defines it. The domain of a partial function f is written $\delta(f)$.

Two partial states are *compatible* exactly if they agree on the intersection of their domains:

$$\sigma_0 \smile \sigma_1 = \forall l \in \delta(\sigma_0) \cap \delta(\sigma_1) \cdot \sigma_0(l) = \sigma_1(l) \quad .$$

Using the pun between states and their graphs, $\sigma_0 \smile \sigma_1$ exactly if $\sigma_0 \cup \sigma_1$ is a state.

For location l and value v, let $l \mapsto v$ be the state such that $\delta(l \mapsto v) = \{l\}$ and $(l \mapsto v)(l) = v$. Using the pun between states and their graphs we have $(l \mapsto v) = \{(l, v)\}$.

[1]The notation $\{i, ..k\}$ means $\{j \in \mathbb{Z} \mid i \le j < k\}$. We assume that $\{0, ..\, \mathrm{length}(a)\} \subseteq V$, for all arrays a.

If σ_0 and σ_1 are states, then $\sigma_0 \triangleright \sigma_1$ is a state such that

$$\delta(\sigma_0 \triangleright \sigma_1) = \delta(\sigma_0) \cup \delta(\sigma_1)$$

$$(\sigma_0 \triangleright \sigma_1)\,(l) = \begin{cases} \sigma_0(l) & \text{if } l \in \delta(\sigma_0) \\ \sigma_1(l) & \text{if } l \notin \delta(\sigma_0) \text{ and } l \in \delta(\sigma_1) \end{cases} \quad .$$

3. EXPRESSIONS

We use two semantic functions for expressions: For lvalue expressions E, we have $[E]_@ \subseteq \Sigma \times (O \cup L \cup A \cup \{\bigstar\})$. For all expressions E of primitive type (int, real etc.), we have $[E]_\$ \subseteq \Sigma \times (V \cup \{\bigstar\})$.

If (σ, x) is in $[E]_@$or $[E]_\$$, then evaluating E in any state $\sigma' \supseteq \sigma$, may yield result x, while reading *exactly* the locations in $\delta(\sigma)$. For example, $[\mathsf{a}[\mathsf{i}]]_@$ contains $(l_0 \mapsto 0, l_1)$, where l_0 is the location of i and l_1 is the location of the first element of array a; it also contains $(l_0 \mapsto -1, \bigstar)$; it does not contain $(l_0 \mapsto 0 \cup l_1 \mapsto 13, l_1)$, as location l_1 is not accessed in computing the location. Now $[\mathsf{a}[\mathsf{i}]]_\$$ contains $(l_0 \mapsto 0 \cup l_1 \mapsto 13, 13)$ as well as $(l_0 \mapsto -1, \bigstar)$.

The defining equations for $[]_@$ and $[]_\$$ are quite straightforward; I provide only a few examples here.[2]

$$\begin{aligned}
[n]_@ &= \{(\emptyset, \mathrm{global}(n))\}\text{, where } n \in \delta(\mathrm{global}) \\
[E.n]_@ &= \{(\sigma, o) \in [E]_@ \cdot (\sigma, \mathrm{field}(o, n))\} \\
[E[F]]_@ &= \{(\sigma_0, a) \in [E]_@, (\sigma_1, i) \in [F]_\$ \mid \sigma_0 \smile \sigma_1 \cdot \\
&\qquad (\sigma_0 \cup \sigma_1, \mathrm{index}(a, i))\} \\
[E]_\$ &= \{(\sigma, l) \in [E]_@, v \in V_E \mid l \in L \wedge \sigma \smile (l \mapsto v) \cdot \\
&\qquad (\sigma \cup (l \mapsto v)\,, v)\} \cup \{(\sigma, \bigstar) \in [E]_@ \cdot (\sigma, \bigstar)\}
\end{aligned}$$

The last equation applies only when E is an lvalue expression of some primitive type; V_E is the subset of V appropriate to E's type.

Nondeterministic expressions can be handled by this approach, but not expressions that change the state.

4. COMMANDS

The meaning of each command C is a set of traces (or sequences) of actions $[C]_!$. We start with a discussion of traces.

4.1. Traces

Given an alphabet of symbols P, P^* is the set of all finite traces over P, while P^ω is the set of infinite traces. Let $P^\infty = P^* \cup P^\omega$. We view traces as functions from initial segments of $\mathbb{N}$. The length $\#s$ of a trace s is $|\delta(s)|$.

The catenation $s; t$ of traces s and t in P^∞ is standard; in particular, $s; t = s$ if $s \in P^\omega$.

[2]The notation $\{v \in S \mid P \cdot E\}$ means the set that contains exactly those things that equal E for some value $v \in S$ such that P. For example $\{n \in \mathbb{N} \mid n \text{ is prime} \cdot n^2\}$ is the set of squares of prime natural numbers: $\{4, 9, 25, 49, \cdots\}$. $\{v \in S \cdot E\}$ means $\{v \in S \mid \mathit{true} \cdot E\}$ and $\{v \in S \mid P\}$ means $\{v \in S \mid P \cdot v\}$.

$$[E_0 := E_1]_! = \{(\sigma_0, l) \in [E_0]_@, (\sigma_1, v) \in [E_1]_\$ \mid \sigma_0 \smile \sigma_1 \wedge l \neq \bigstar \wedge v \neq \bigstar \cdot \mathbf{start}(\sigma_0 \cup \sigma_1, \{l\}); \mathbf{fin}(l \mapsto v, \delta(\sigma_0) \cup \delta(\sigma_1))\} \tag{2}$$

$$\cup \{(\sigma_0, l) \in [E_0]_@, (\sigma_1, v) \in [E_1]_\$ \mid \sigma_0 \smile \sigma_1 \wedge (l = \bigstar \vee v = \bigstar) \cdot \mathbf{chaos}(\sigma_0 \cup \sigma_1)\}$$

$$[C_0\ C_1]_! = [C_0]_! \,; [C_1]_! \tag{3}$$

$$\mathrm{filter}(E) = \big\{\sigma \mid (\sigma, \mathsf{true}) \in [E]_\$ \cdot \mathbf{start}(\sigma, \emptyset), \mathbf{fin}(\emptyset, \delta(\sigma))\big\} \cup \big\{\sigma \mid (\sigma, \bigstar) \in [E]_\$ \cdot \mathbf{chaos}(\sigma)\big\}$$

$$[(\mathbf{if}\ E\ C_0\ \mathbf{else}\ C_1\ \mathbf{if})]_! = \mathrm{filter}(E); [C_0]_! \cup \mathrm{filter}(\neg E); [C_1]_! \tag{4}$$

$$[(\mathbf{wh}\ E\ C\ \mathbf{wh})]_! = \big((\mathrm{filter}(E); [C]_!)^* \,; \mathrm{filter}(\neg E)\big) \cup (\mathrm{filter}(E); [C]_!)^\omega \tag{5}$$

$$[(\mathbf{co}\ C_0 \parallel C_1\ \mathbf{co})]_! = [C_0]_! \parallel [C_1]_! \tag{6}$$

$$\mathrm{enter}(k, E) = (\mathbf{try}(\{k\})^\infty; \mathbf{acq}(k); \mathrm{filter}(\neg E); \mathbf{rel}(k))^\infty \,; \mathbf{try}(\{k\})^\infty; \mathbf{acq}(k); \mathrm{filter}(E)$$

$$\mathrm{enter}(k) = \mathbf{try}(\{k\})^\infty; \mathbf{acq}(k)$$

$$[(\mathbf{with}\ E_0\ \mathbf{when}\ E_1\ C\ \mathbf{with})]_! = \bigcup_{(\sigma,k) \in [E_0]_@ \mid k \neq \bigstar} \mathbf{start}(\sigma, \emptyset); \mathbf{fin}(\emptyset, \delta(\sigma)); \mathrm{enter}(k, E_1); [C]_! \,; \mathbf{rel}(k) \tag{7}$$

$$\cup \{(\sigma, k) \in [E_0]_@ \mid k = \bigstar \cdot \mathbf{chaos}(\sigma)\}$$

$$[E.n()]_! = \bigcup_{(\sigma,o) \in [E]_@ \mid o \neq \bigstar} \mathbf{start}(\sigma, \emptyset); \mathbf{fin}(\emptyset, \delta(\sigma)); \mathbf{rel}(\mathfrak{a}(o, n)); \mathrm{enter}(\mathfrak{d}(o, n)) \tag{8}$$

$$\cup \{(\sigma, k) \in [E_0]_@ \mid k = \bigstar \cdot \mathbf{chaos}(\sigma)\}$$

$$[(\mathbf{accept}\ n()\ C\ \mathbf{accept})_o]_! = \mathrm{enter}(\mathfrak{a}(o, n)); [C]_!; \mathbf{rel}(\mathfrak{d}(o, n)) \tag{9}$$

Fig. 1. Semantics for commands

A trace set S is any subset of P^∞. For trace sets S and T: $S;T$ is the elementwise catenation of trace sets, S^* is the Kleene closure of S, S^ω is the infinite catenation of S with itself, and $S^\infty = S^* \cup S^\omega$.

For two traces s and t, we define the fair merge $s \parallel t$ of the traces as the trace set generated by breaking s and t into any number of finite pieces that are then interleaved. The fair merge of two trace sets S and T is $S \parallel T = \bigcup_{s \in S, t \in T} s \parallel t$.

As usual in formal language theory, I will pun between a symbol p, trace $\langle p \rangle$, and trace set $\{\langle p \rangle\}$, when there is no ambiguity; for example $p; q = \langle p \rangle \,; \langle q \rangle$ and $p^* = \{\langle p \rangle\}^*$.

4.2. Actions

Our symbol set P is a set of *actions*, borrowed in the main from [1]. We describe actions informally here and formally in Section 4.6. In each case σ is a state.

- $\mathbf{start}(\sigma, W)$, where W is a set of locations. This action is enabled in states compatible with σ. It marks the domain of σ as "being read" and marks all location in W as "being written".
- $\mathbf{fin}(\sigma, R)$, where R is a set of locations. This action is always enabled. It updates the state according to σ. The domain of σ is unmarked as "being written" and the set R is unmarked as "being read".
- $\mathbf{chaos}(\sigma)$.This action is enabled in states compatible with σ. It is completely nondeterministic.

4.3. Command semantics

Now we are ready to give the trace sets associated with sequential commands. These are given in Figure 1.

A crucial command is assignment; see (2). If either expression evaluates to $\bigstar$, the trace simply a **chaos** action. For other states, the trace consists of a **start** action and a **fin** action with appropriate marking and unmarking of locations.

Sequential composition (3) is reflected by catenation of trace sets. 'If' commands (4) and 'while' commands (5) are defined with the help of the filter function. Parallel composition is simply a fair merge of the trace sets (6).

4.4. Lock actions

In HARPO/L objects implementing the Lock interface are locks for conditional critical sections. At run-time each lock has two states: locked and unlocked. The following actions affect locks:

- $\mathbf{try}(K)$ is a failed attempt to acquire a lock in set K. This action is enabled when all locks in K are locked. It has no effect.
- $\mathbf{acq}(k)$ is a successful acquisition of lock k. This action is enabled when lock k is unlocked; it locks k.
- $\mathbf{rel}(k)$ is the release of lock k. This action is always enabled; it unlocks k.

$$(\sigma_0, R_0, W_0, K_0, \text{true}) \overset{\mathbf{start}(\sigma,W)}{\rightarrowtail} \begin{cases} (\sigma_0, \delta(\sigma) \uplus R_0, W_0 \cup W, K_0, \text{true}) \\ \quad \text{if } \sigma \smile \sigma_0 \text{ and } \delta(\sigma) \cap W_0 = \emptyset \text{ and } W \cap (W_0 \cup R_0) = \emptyset \\ (\sigma_1, R_1, W_1, K_1, ok_1) \\ \quad \text{if } \sigma \smile \sigma_0 \text{ and (} \delta(\sigma) \cap W_0 \neq \emptyset \text{ or } W \cap (W_0 \cup R_0) \neq \emptyset \text{)} \end{cases} \tag{10}$$

$$(\sigma_0, R_0, W_0, K_0, \text{true}) \overset{\mathbf{fin}(\sigma,R)}{\rightarrowtail} (\sigma \triangleright \sigma_0, R_0 - R, K_0, W_0 - \delta(\sigma), \text{true}) \tag{11}$$

$$(\sigma_0, R_0, W_0, K_0, \text{true}) \overset{\mathbf{chaos}(\sigma)}{\rightarrowtail} (\sigma_1, R_1, W_1, K_1, ok_1) \qquad \text{if } \sigma \smile \sigma_0$$

$$(\sigma_0, R_0, W_0, K_0, \text{true}) \overset{\mathbf{try}(K)}{\rightarrowtail} (\sigma_0, R_0, W_0, K_0, \text{true}) \qquad \text{if } K \subseteq K_0$$

$$(\sigma_0, R_0, W_0, K_0, \text{true}) \overset{\mathbf{acq}(k)}{\rightarrowtail} (\sigma_0, R_0, W_0, K_0 \cup \{k\}, \text{true}) \qquad \text{if } k \notin K_0$$

$$(\sigma_0, R_0, W_0, K_0, \text{true}) \overset{\mathbf{rel}(k)}{\rightarrowtail} (\sigma_0, R_0, W_0, K_0 - \{k\}, \text{true})$$

$$(\sigma_0, R_0, W_0, K_0, \text{false}) \overset{p}{\rightarrowtail} (\sigma_1, R_1, W_1, K_1, ok_1)$$

Fig. 2. Semantics for actions

4.5. Interprocess synchronization

HARPO/L has two mechanisms for interprocess synchronization: conditional critical sections and rendezvous.

Conditional critical sections, also known as 'with' commands, are shown in (7). Except for the need to compute the location of the lock object, this is essentially the same as in [1], [3] and [4].

A rendezvous is an asymmetric communication mechanism [5]. Simultaneously the client executes a method call $E.n()$ while, the server executes an 'accept' command, which in its simplest form is[3] $(\mathbf{accept}\ n()\ C)_o$. The server must wait at an accept until there is a client ready to rendezvous. The client must wait until the server is ready to execute the 'accept' command. The semantics for this simplest form of rendezvous is shown in (8–9). There are two initially locked locks associated with the method named n in object o: $\mathfrak{a}(o, n)$ is used for the server to wait for the client, while $\mathfrak{d}(o, n)$ is used for the client to wait for the server.

In their full generality, 'accept' commands are fairly complex. Space does not permit a full exposition in the present paper, although the details are worked out in the full semantic description for the language. The full 'accept' command can generalize the simplest form in three ways.

- 'Accept' commands can have both input and output parameters. To handle these, a protocol using up to five locks is used: server waits for client, client waits before copying input parameters, server waits before executing the method body, client waits before copying output parameters, server waits until parameters are copied.

[3]Recall that classes have been instantiated to objects at an earlier stage. We assume that, as part of instantiation, each 'accept' command has been tagged with the object o that it appears in.

- 'Accept' commands can have multiple branches, so that multiple methods can be implemented by the same 'accept' command. In that case the server first waits on all $\mathfrak{a}$ locks associated with the various methods. This is why the 'try' action has a set of locks.
- 'Accept' commands can have boolean guard expressions. These are evaluated first and govern which $\mathfrak{a}$ locks are waited on.

4.6. The semantics of traces

We supply a meaning to each trace by means of an operational semantics. In effect we define a nondeterministic state machine for traces. Each trace then defines a binary relation on states of the machine. We call the states of this machine *configurations*, as the word 'state' is already in use for the data state. Each configuration consists is a 5-tuple (σ, R, W, K, ok) where σ is a total state, R is a multiset of locations currently being read, W is a set of locations currently being written, K is a set of locks currently locked, and ok is a boolean indicating that the computation has not gone wrong. The R, W, and ok fields are purely fictions, there is no need to actually represent them at run-time.

Each action p defines a binary relation $\overset{p}{\rightarrowtail}$ on configurations. The meaning of each action is given in Fig 2. For a given configuration x and action p there are three possibilities.

- There exists no y such that $x \overset{p}{\rightarrowtail} y$. Then we say that the action is not enabled.
- For all y, $x \overset{p}{\rightarrowtail} y$. In this case the computation has gone wrong and anything can happen.
- Somewhere in between.

From (10), you can see that **start** actions are only enabled in configurations where the state is compatible with the partial state of the action. Then there are two possibilities. If the action requires a read of a location currently being written or a write of a location currently being read or written, then the computation has gone wrong and any configuration could be next. Otherwise, the locations are marked and the computation proceeds. The **fin** actions (11) are a bit simpler, they simply modify part of the state and release read and write claims made in the corresponding start operation. The other actions are straight-forward.

We can extend the relation to finite sequences as the least relation such that $x \overset{\varepsilon}{\rightarrowtail} x$ and $x \overset{ps}{\rightarrowtail} z$ if there exists y such that $x \overset{p}{\rightarrowtail} y$ and $y \overset{s}{\rightarrowtail} z$. We extend it to infinite sequences by saying that $x \overset{s}{\rightarrowtail} z$ exactly if there is an infinite computation

$$x \overset{s(0)}{\rightarrowtail} y_1 \overset{s(1)}{\rightarrowtail} y_2 \overset{s(2)}{\rightarrowtail} y_3 \cdots \quad .$$

Now for a trace set S, $x \overset{S}{\rightarrowtail} y$ exactly if $\exists s \in S \cdot x \overset{s}{\rightarrowtail} y$. Each command C gives a relation $\overset{[C]_!}{\rightarrowtail}$.

5. RELATED AND FUTURE WORK

The work presented here is strongly based on that in [1]. I note the following differences:

- HARPO/L is a more complex language than that in [1]. It has objects, rendezvous, and expressions that can go wrong.
- HARPO/L allows concurrent read operations. In [1], partial states are used in the configuration with the missing part of the domain indicating which locations are marked as unusable. I could not adopt this approach as there are two ways in which locations can be unusable. This leads to the use of the R and W (multi-)sets.
- I used a different approach to computations gone wrong. Whereas [1] uses a special **wrong** value to indicate computations gone wrong. I use chaotic computations, which lead to a simpler formulation.
- I used a different approach to infinite computations. Whereas [1] uses a special trace $\perp$ to indicate infinite computations. I use infinite traces. This leads to a simpler formalization.
- In [1] the trace is part of the configuration. This seemed unnecessary to me, as only the first action is actually used.
- I used a **try** action that tries multiple locks. This change is not necessary, the same effect can be had with interleaving actions that each try only one lock. The motivation is a cleaner approach to accept statements with multiple branches.

The work here shows that with only moderate extensions the approach of [1] can be extended to a more complex language.

Two closely approaches have been developed by Brookes. In [3], actions indicate individual reads and writes. In [4], state changes from a number of statements are combined. Thus purely sequential parts of the computation are given essentially the semantics that they would be given in a two-state sequential model. This latter approach is intriguing and may form the basis of future work on the HARPO/L semantics.

Future work will include an investigation of equivalence of traces and how it applies to the safety of optimizations. Two traces s and t are equivalent, $s \equiv t$, iff for all x, y, and u, $(x \overset{s||u}{\rightarrowtail} y) = (x \overset{t||u}{\rightarrowtail} y)$. This notion of equivalence can be applied to program optimization as follows. Let C and D be two commands, the C is refined by D, written $C \sqsubseteq D$, iff $\forall s \in [D]_! \cdot \exists t \in [C]_! \cdot s \equiv t$. Then a source level program optimization f is safe on C if $C \sqsubseteq f(C)$. A similar idea can be applied to low-level optimizations, as long as the effect on the trace sets is clear. This notion of refinement is entirely local and does not consider the context of the command.

The intermediate representation used in our compiler is a form of dataflow graph. By giving a grainless semantics to these graphs, we may be able to at least informally verify the stage of the compiler that produces dataflow graphs.

6. REFERENCES

[1] John C. Reynolds, "Towards a grainless semantics for shared-variable concurrency," in *Proc. 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, 2005, vol. 3328 of *Lecture Notes in Computer Science*, pp. 35–48.

[2] Theodore S. Norvell, "HARPO/L: A language for hardware/software codesign.," in *Newfoundland Electrical and Computer Engineering Conference (NECEC)*, 2008.

[3] Stephen Brookes, "A semantics for concurrent separation logic," *Theoretical Computer Science*, vol. 375, no. 1–3, pp. 227–270, May 2007.

[4] Stephen Brookes, "A grainless semantics for parallel programs with shared mutable data," *Electronic Notes in Theoretical Computer Science*, vol. 155, pp. 277–307, 2006, Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

[5] Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard, "Rationale for the design of the Ada programming language," *SIGPLAN Not.*, vol. 14, no. 6b, pp. 1–261, 1979.