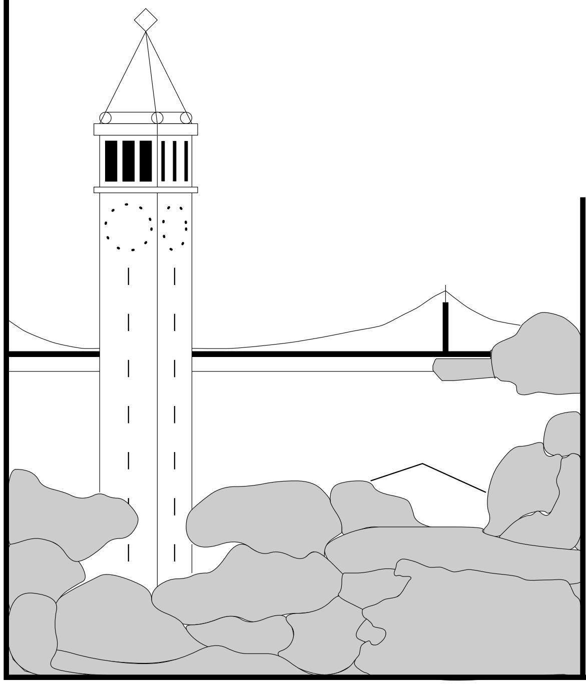
Barrier Inference

David Gay



Report No. UCB/CSD-97-965

July 1997

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Barrier Inference

David Gay*
EECS Department
University of California, Berkeley
dgay@cs.berkeley.edu

Abstract

Many parallel programs are written in SPMD style, i.e. by running the same sequential program on all processes. SPMD programs include synchronization, but it is easy to write incorrect synchronization patterns. We propose a system that verifies a program's synchronization pattern. We also propose language features to make the synchronization pattern more explicit and easily checked. We have implemented a prototype of our system for Split-C and successfully verified the synchronization structure of realistic programs.

1 Introduction

Explicitly-parallel programming—where the programmer specifies the parallelism in a computation—is arguably the most widely used parallel programming paradigm. Despite many years of practical experience, there has been little work on the static semantics of explicitly-parallel programming languages. We propose a static semantics for global synchronization that guarantees an explicitly parallel program has no global synchronization errors. Our proposal is based on a formalization of widespread programming practices. We have proven the soundness of our method and implemented a prototype system. Experimental evidence gathered from testing our system on a benchmark suite supports our hypothesis that the global synchronization structure of realistic programs can be formalized and automatically verified.

Our system was developed in the context of a distributed memory, shared address space programming language (Split-C, an SPMD language developed at Berkeley [5]), but we found it equally applicable to checking the synchronization structure of shared memory, shared address space parallel programs; our method can show the synchronization correctness of the SPLASH-2 [25] benchmarks. We expect a similar result should hold for pure message passing programs as well, but such programs may not rely on global synchronization to the same degree as programs written using a shared address space. Note, however, that standard message passing libraries such as MPI [20] include global synchronization primitives.

1.1 Global Synchronization

A simple and popular parallel programming model is SPMD (for Single Program, Multiple Data). SPMD programs are explicitly-parallel programs written in sequential languages extended with communication and synchronization primitives. A typical SPMD code skeleton is

```
work1();
barrier;
work2();
```

*This material is based in part upon work supported by DARPA contract F30602-95-C-0136.

```
if (random()) barrier;
work1(); barrier;
work2();
```

(a) processes left behind

```
while (random()) barrier;
work1(); barrier;
work2(); barrier;
work3();
```

(b) processes "trapped" in a loop

```
if (random()) barrier else broadcast;
```

(c) conflicting barrier/broadcast

```
a <- random();
if (a) barrier;         (*)
x <- x + 1;
if (not a) barrier;     (*)
```

(d) correct but not structurally correct

```
if (x) barrier else work()
```

(e) correct if processes agree on `x`'s value

```
i <- 0;
while i < 10
  (if (i = 1) barrier;
   i <- i + 1);
barrier
```

(f) correct loop

```
if (random())
  (barrier; barrier)
else
  (work1(); barrier; work2(); barrier)
```

(g) if with matching barriers

```
i <- 0
if (random())
  (while (i < 10) (barrier; i <- i + 1))
else
  (j <- i + 10;
   while (j < 20) (work1(); barrier; j <- j + 1))
```

(h) structurally correct but not verifiable

Figure 1: Examples

```
barrier;
work3();
```

where `barrier` is an operation that causes a process to block until all other processes have also reached a `barrier`. In SPMD execution, all processes execute a copy of the program independently. In this example, the `barrier`s serve to guarantee that, e.g., all processes are done with `work1()` before proceeding to `work2()`. The only synchronization is at the `barrier`s—processes execute `workn()` asynchronously.

While conceptually simple, the combination of asynchronous execution and explicit global synchronization introduces subtle issues of program structure and correctness. Figure 1 gives examples illustrating correct and incorrect synchronization patterns. In these examples, `random()` returns a different value in every process (causing different branch decisions in different processes) and `workn()` is a function call that does no synchronization. In all of the examples `barrier`s are executed conditionally; we have observed that almost all SPMD programs have conditional synchronization.

There are two basic forms of incorrect synchronization. In Figure 1a, processes execute different numbers of `barrier`s, causing the program to "hang" when some processes terminate while others wait at a `barrier`. The same problem occurs in loops containing `barrier`s if processes execute differing numbers of iterations (Figure 1b). The second problem is illustrated by Figure 1c, where some processes execute a `barrier` while others execute a `broadcast`. In SPMD languages simultaneously executing different synchronization operations causes a runtime error (or, in some implementations, undefined behavior).

Even correct SPMD synchronization can be subtle. Figure 1e is correct, provided that the values of variable `x` (which is a *replicated* variable, i.e. each process has a variable `x` local to the process) is the same in all processes. This pattern—conditional synchronization where the program's design guarantees processes make the same branch decisions—is ubiquitous in SPMD programs. Figure 1f gives a more complex example illustrating the same point. However, in correct programs processes need not always make the same branch decisions, as Figures 1d, g, and h show.

1.2 Synchronization Verification

Figure 1e shows that an important component of understanding synchronization behavior is knowing which replicated variables must have the same value in all processes: We call such variables *single-valued*[1]. Replicated variables that may have different values in different processes are *multi-valued*. In practice, SPMD programmers use synchronization in a highly structured way. All SPMD programs we have seen observe the following notion of synchronization correctness.

Definition 1.1 (Structural Correctness) An expression is structurally correct if all subexpressions e satisfy the following: Let V be the set of single-valued variables on entry to e. If processes begin execution of e in environments that agree on the values of V and all processes terminate (i.e., no process loops), then all processes execute the same sequence of synchronization operations.

It is easy to check that Figure 1f, g, and h are all structurally correct and that Figure 1e is structurally correct assuming `x` is single-valued. Figure 1d is an example of a program that has no synchronization errors but is not structurally correct (because of the expressions marked $(*)$).

1.3 Barrier Inference

We have developed a static semantics that verifies that a program has structurally correct synchronization. Since `barrier`s are the most common form of SPMD synchronization, we call this process *barrier inference*. Statically checking synchronization behavior guarantees that programs never fail by "hanging" or executing conflicting synchronization operations. SPMD programmers do make such mistakes[2], and our techniques eliminate this class of bugs. Equally important, our method makes explicit the heretofore implicit assumptions about single-valued variables in SPMD programs. In our experience, this extra information is extremely useful for understanding SPMD programs written by others.

There are structurally correct programs that our system cannot verify, such as Figure 1h. Intuitively, the problem with this example is that although both branches execute the same number of barriers, our system can only infer that the branches each execute some unknown number of barriers and cannot tell that these numbers are the same. In contrast, our system has no difficulty with Figure 1g, where the system can infer that both branches execute exactly two barriers. While we have seen examples similar to Figure 1g, we have seen no programs with the structure of Figure 1h.

We present our barrier inference algorithm, which statically verifies the correctness of an SPMD program's synchronization behavior (Section 3), along with a proof of its soundness (Section 3.1). We also propose language features that make the synchronization structure of SPMD programs explicit (Section 4.1). We have implemented a prototype system to validate the algorithm and to empirically study the proposed language features. We tested the prototype on a substantial number of programs written in Split-C (Section 5). Experience with our implementation is positive; the system successfully checks the benchmarks with a few minor modifications to the programs, including one to correct a bug detected by our system. We have also examined the Splash-2 benchmarks [25] by hand and found that all but one can be checked with our system (Section 5.2). These experiments were for medium-size programs; we believe that static verification of synchronization is especially important for larger systems because these are not amenable to manual verification, and also for higher-order languages (e.g. parallel object-oriented languages) where control-flow is less explicit.

[1] A formal definition of single-valued variables is subtle; see Section 3.1.

[2] It is difficult to provide direct evidence for this claim, but we have committed such programming mistakes ourselves and found them in existing, presumably debugged, programs.

2 The Language

We present our system using $\mathcal{L}$, a small procedural language extended with three parallel operations: `barrier`, `broadcast` (which is like `barrier` except a distinguished value is sent to all processes), and `communicate` (which allows asynchronous communication). As our interest is in synchronization operations such as `barrier` and `broadcast`, we leave the semantics of `communicate` unspecified. The grammar for $\mathcal{L}$ is:

$$
\begin{array}{lcl}
\texttt{Expr} & ::= & \texttt{i} \\
 & | & \texttt{id} \\
 & | & \texttt{barrier} \\
 & | & \texttt{broadcast} \\
 & | & \texttt{communicate} \\
 & | & \texttt{id(Expr}, \ldots, \texttt{Expr)} \\
 & | & \texttt{id} \leftarrow \texttt{Expr} \\
 & | & \texttt{if Expr Expr else Expr} \\
 & | & \texttt{Expr; Expr} \\
 & | & \texttt{let id in Expr} \\
 & | & \texttt{letrec id(id}, \ldots, \texttt{id)} = \texttt{Expr in Expr}
\end{array}
$$

All values in $\mathcal{L}$ are integers and all variables are replicated. A `let` introduces a new integer variable and a `letrec` introduces a potentially recursive function definition; the other expressions are also standard. There are some predefined functions, such as $+$, which are all mathematical functions, i.e. their result depends solely on the value of their arguments. In examples we write `while` $\texttt{e}_1$ $\texttt{e}_2$ as shorthand for

$$\texttt{letrec f()} = \texttt{if } \texttt{e}_1 \ (\texttt{e}_2\texttt{; f())} \texttt{ else 0 in f()}$$

This sparse language is sufficient to illustrate the novel aspects of our techniques. In Section 4.3 we discuss extensions to the C and FORTRAN-based languages used in practice. Figure 2 gives a simple rewrite semantics for $\mathcal{L}$ in a variation of continuation-passing style (CPS). The computation of a single process is a sequence of steps:

$$\texttt{State} \leadsto \texttt{State}$$

where a state $\texttt{FunEnv} \times \texttt{Env} \times \texttt{Cont} \times \texttt{Expr}$ consists of an expression e to be evaluated, environments for the variables and function names in scope at e, and the computation to perform after evaluating e (a continuation). Readers familiar with CPS semantics will note that this CPS semantics is non-standard, because a continuation is a function that returns only the next state in the computation, rather than the final answer of the entire computation. This modification exposes intermediate states of the computation, which is needed to define the semantics of `barrier` and `broadcast`.

The semantics of $\mathcal{L}$ model synchronization structure, but not the details of the communication primitives. The synchronization primitives, `barrier` and `broadcast`, are the only operations that require global interaction. For `barrier`, once all processes reach a `barrier` each process proceeds with its continuation. The rule for `broadcast` is identical. The values returned by the communication operations are predicted by an $oracle()$ function. The only place where the communicated value is important is in `broadcast`: it returns the same value in all processes, but the actual value is not important for synchronization verification. The `barrier` operation does not communicate any values, so its result is always 0 (an arbitrary choice).

A few other comments on details of the semantics are necessary. For simplicity, we assume that variables and functions are given unique names (i.e., no names hide names in outer scopes). This property can always be enforced by suitably renaming variables. Define $FF(f)$ as the set of function names in scope at f's definition; ; $FV(f)$ is the set of identifiers (other than f's formal parameters) in scope at f's definition. Figure 2 uses several operations on environments. The set $dom(E)$ is the domain of E. The environment $E|V$ is E with the domain

$$\begin{array}{ll}
F & \mathtt{FunEnv} = \mathtt{FunctionName} \rightarrow \mathtt{FunctionDefinition} \\
E & \mathtt{Env} = \mathtt{Var} \rightarrow \mathcal{N} \\
C & \mathtt{Cont} = \mathtt{Env} \times \mathcal{N} \rightarrow \mathtt{State} \\
 & \mathtt{State} = \mathtt{FunEnv} \times \mathtt{Env} \times \mathtt{Cont} \times \mathtt{Expression} + \mathtt{Env} \times \mathcal{N}
\end{array}$$

$$\begin{array}{rcl}
\langle F, E, C, i \rangle & \rightsquigarrow & C\ (E, i) \\[1ex]
\langle F, E, C, x \rangle & \rightsquigarrow & C\ (E, E(x)) \\[1ex]
\langle F, E, C, communicate \rangle & \rightsquigarrow & C\ (E, oracle()) \\[1ex]
\langle F, E, C_0, f(Expr_1, \ldots, Expr_n) \rangle & \rightsquigarrow & \langle F, E, C_1, Expr_1 \rangle \text{ where} \\
 & & F(f) = \ f(x_1, \ldots, x_n) = Expr \\
 & & C_1 = \lambda E_2, v_1.\langle F, E_2, C_2, Expr_2 \rangle \\
 & & \ldots \\
 & & C_{n-1} = \lambda E_n, v_{n-1}.\langle F, E_n, C_n, Expr_n \rangle \\
 & & C_n = \lambda E_{n+1}, v_n.\langle F|\ FF(f), E_0, C', Expr \rangle \\
 & & E_0 = (E_{n+1}|FV(f))[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] \\
 & & C' = \lambda E', v.C_0\ ((E_{n+1}//FV(f) + E'//\{x_1, \ldots, x_n\}), v) \\[1ex]
\langle F, E, C_0, p(Expr_1, \ldots, Expr_n) \rangle & \rightsquigarrow & \langle F, E, C_1, Expr_1 \rangle \text{ where} \\
 & & p \text{ is a primitive} \\
 & & C_1 = \lambda E_2, v_1.\langle F, E_2, C_2, Expr_2 \rangle \\
 & & \ldots \\
 & & C_{n-1} = \lambda E_n, v_{n-1}.\langle F, E_n, C_n, Expr_n \rangle \\
 & & C_n = \lambda E_{n+1}, v_n.C_0(E_{n+1}, p\,(v_1, \ldots, v_n)) \\[1ex]
\langle F, E, C, x \leftarrow Expr \rangle & \rightsquigarrow & \langle F, E,\ \lambda E', v.C\ (E'[x \leftarrow v], v),\ Expr \rangle \\[1ex]
\langle F, E, C, if\ Expr_1\ Expr_2\ else\ Expr_3 \rangle & \rightsquigarrow & \langle F, E, C_0, Expr_1 \rangle \text{ where} \\
 & & C_0 = \lambda E', v.\langle F, E', C, \text{ if } v = 0 \text{ then } Expr_2 \text{ else } Expr_3 \rangle \\[1ex]
\langle F, E, C, Expr_1; Expr_2 \rangle & \rightsquigarrow & \langle F, E,\ \lambda E', v.\langle F, E', C, Expr_2 \rangle,\ Expr_1 \rangle \\[1ex]
\langle F, E, C, let\ x\ in\ Expr \rangle & \rightsquigarrow & \langle F, E[x \leftarrow 0],\ \lambda E', v.C\ (E'//\{x\}, v),\ Expr \rangle \\[1ex]
\langle F, E, C, letrec\ f(x_1, \ldots, x_n) = Expr_1\ in\ Expr_2 \rangle & \rightsquigarrow & \langle F[f \leftarrow f(x_1, \ldots, x_n) = Expr_1], E, C, Expr_2 \rangle \\
 & & FF(f) = dom(F), \quad FV(f) = dom(E) \\[1ex]
[\langle F_1, E_1,\ C_1,\ barrier \rangle, \ldots, \langle F_n, E_n,\ C_n,\ barrier \rangle] & \rightsquigarrow & [C_1\ (E_1, 0), \ldots, C_n\ (E_n, 0)] \\[1ex]
[\langle F_1, E_1,\ C_1,\ broadcast \rangle, \ldots, \langle F_n, E_n,\ C_n,\ broadcast \rangle] & \rightsquigarrow & [C_1\ (E_1, v), \ldots, C_n\ (E_n, v)] \text{ where } v = oracle()
\end{array}$$

Figure 2: Semantics for $\mathcal{L}$

restricted to variables V. The environment $E//V$ is E with variables V removed; i.e., $E|(dom(E) - V)$. The environment $E_1 + E_2$ is the combination of two environments E_1 and E_2 with disjoint domains.

The result of a (terminating) sequence of rewrites is an environment recording the final state of the program and an integer result. The computation of n processes executing in parallel is a sequence of steps:

$$\mathtt{State^n} \rightsquigarrow \mathtt{State^n}$$

The transitions for vectors of states include the synchronization rules for `barrier` and `broadcast`, plus a general rule for interleaving the transitions of individual processes:

$$[\mathtt{S_1}, \ldots, \mathtt{S_{i-1}}, \mathtt{S_i}, \mathtt{S_{i+1}}, \ldots \mathtt{S_n}] \rightsquigarrow [\mathtt{S_1}, \ldots, \mathtt{S_{i-1}}, \mathtt{S'_i}, \mathtt{S_{i+1}}, \ldots \mathtt{S_n}]$$

whenever $\mathtt{S_i} \rightsquigarrow \mathtt{S'_i}$. Let I be the initial continuation $\lambda E, v.(E, v)$. The evaluation of an expression e on n processors is

$$[\langle\{\emptyset, \mathtt{pid} = \mathtt{1}\},\ \mathtt{I},\ \mathtt{e}\rangle, \ldots n\ times \ldots, \langle\{\emptyset, \mathtt{pid} = \mathtt{n}\},\ \mathtt{I},\ \mathtt{e}\rangle] \overset{*}{\rightsquigarrow} [(\mathtt{E_1}, \mathtt{i_1}), \ldots, (\mathtt{E_n}, \mathtt{i_n})]$$

The initial environment of each process contains a *process id* in the variable `pid`. This value distinguishes one process from another.

If all processes halt with a final environment and integer value then that run is successful. A run is unsuccessful if (1) processes execute a different number of `barriers` (Figures 1a and 1b), (2) some processes reach a `barrier` at the same time others reach a `broadcast` (Figure 1c), or (3) one or more processes loop. Our methods are capable of statically checking realistic programs for (1) and (2).

3 Barrier Inference

The rules of our inference system model two aspects of SPMD computation. The first aspect is the sequence of `barriers` and `broadcasts` executed in evaluating an expression e. The rules associate an abstract synchronization sequence with e:

$$\mathcal{S} = \{\perp, f\} \cup \{b, r\}^*$$

A sequence value $s \in \{b, r\}^*$ means every process executes exactly the sequence s of `barriers` (b) and `broadcasts` (r). A sequence value f means every process executes the same sequence of `barriers` and `broadcasts`, but the exact sequence is unknown. The sequence value $\perp$ means no process executes the expression. It is possible to assign an element of $\mathcal{S}$ to every structurally correct expression. There is an ordering on synchronization sequences:

$$\perp \preceq s \preceq f \quad \text{for any } s \in \{b, r\}^*$$

The second aspect of the inference rules tracks single-valued variables. An abstract environment $\mathtt{AEnv} : \mathtt{Vars} \rightarrow \{+, -\}$ is a mapping from program variables to $+$ (indicating a variable is single-valued) or $-$ (indicating a variable may be multi-valued). There is an ordering $+ \preceq -$.

Analogous to an abstract environment there is an *abstract function environment*, which is a mapping

$$\mathtt{FEnv} : \mathtt{FunctionNames} \rightarrow \{+, -\}^{\mathtt{n}} \times \mathtt{AEnv} \times \{+, -\} \times \mathtt{AEnv} \times \mathcal{S}$$

from function names to function signatures.

Definition 3.1 A function f satisfies a function signature written

$$(\mathtt{a_1}, \ldots, \mathtt{a_n}), \mathtt{A} \rightarrow \mathtt{a}, \mathtt{A'}, \mathtt{s}$$

if the following hold: f has n arguments and its free variables are those in $dom(A) = dom(A')$; processes that begin execution of f in states agreeing on values of the single-valued function arguments in $(\mathtt{a_1}, \ldots, \mathtt{a_n})$ and single-valued variables in $\mathtt{A}$, either diverge or (1) agree on the result if $\mathtt{a} = +$, (2) agree on the value of every single-valued variable in $\mathtt{A'}$, and (3) have executed the same sequence of synchronization operations $\mathtt{s}$.

For example, the signature

$$f : (+, -), \emptyset \to +, \emptyset, \epsilon$$

says that $f(a, b) = f(a, c)$ for all b and c (provided both evaluations terminate) and f executes no synchronization operations. The inference system proves statements of the form

$$\mathtt{B},\ \mathtt{A} \vdash \mathtt{Expr} : \mathtt{a},\ \mathtt{A}',\ \mathtt{s}$$

which is read: Given functions with abstract function environment `B`, if all processes begin the execution of `Expr` with the same values for variables marked single-valued in `A`, then all processes that terminate (1) agree on the values of variables marked single-valued in `A'`, (2) agree on the result if $\mathtt{a} = +$, and (3) have executed the same sequence of synchronization operations `s`. Thus, any such proof shows e's structural correctness (Definition 1.1).

The inference rules are given in Figure 3. In the remainder of this section we discuss the rules, present a soundness result, and illustrate barrier inference with some examples. The [Int] rule is simple; evaluating an integer is single-valued (all processes compute the same integer), does not affect the set of single-valued variables in the environment, and executes no synchronization operations. The [Id] rule is similar; the result is single-valued only if all processes have the same value for the identifier in the environment. A `communicate` is assumed to be multi-valued, as processes may receive different values. When a process needs to communicate a value to all processes, `broadcast` is more efficient than n `communicate` operations, and makes explicit that the result is single-valued[3]. A `barrier` and a `broadcast` are always single-valued and each executes a single synchronization operation. The [Prim] rule says that primitive, side-effect free functions are single-valued if all their arguments are single-valued.

In rule [Fun], actual parameters must be single-valued wherever the function signature requires single-valued arguments (the comparisons $a_i \preceq a'_i$). Similarly, the environment of the call must be single-valued in all variables the signature requires be single-valued. We define $A_1 \preceq A_2$ if $dom(A_1) = dom(A_2)$ and for all $x \in dom(A_1)$ we have $A_1(x) \preceq A_2(x)$.

The conclusion of [Fun] and several of the other rules combine synchronization sequences. The sequence $s_1 \oplus s_2$ is the best description of s_1 followed by s_2:

$$s_1 \oplus s_2 = \begin{cases} s_1 \cdot s_2 & \text{if } s_1, s_2 \in \{b, r\}^* \\ \bot & \text{if } s_1 = \bot \vee s_2 = \bot \\ s_1 \sqcup s_2 & \text{otherwise} \end{cases}$$

where $s_1 \cdot s_2$ is the concatenation of strings s_1 and s_2. The operator $\oplus$ is monotonic in both arguments.

Note the difference between the treatment of primitive and user-defined functions. The result of a primitive function is single-valued if all its arguments are single-valued, which can be thought of as a kind of subtyping rule. Thus, some uses of a primitive function can be single-valued and others not. All the calls to a user-defined function are either single-valued or not, depending on the function's signature in the abstract function environment. This distinction is necessary, because user-defined functions with side-effects can modify single-valued state. We have not found this restriction on user-defined functions to be a problem in practice (see Section 5.1).

The [Assign] rule updates the environment based on the new value of the assigned variable; this reflects the fact that a variable can be single-valued at some points in the program and not at others. The [Let] rule introduces a new variable, which is initially single-valued as it is initialised to 0 in all processes. A new function is introduced into the function environment by the [LetRec] rule. This rule, along with the [Fun] rule, only expresses constraints on the function's signature, but does not specify how it is found. Section 3.2 outlines a method for computing function signatures.

The two rules for `if` are interesting. The rule [If-Single] applies when the predicate is single-valued. All processes take the same branch, but we do not know which branch. In this case a conservative upper bound over the results of both branches suffices.

The rule [If-Multi] applies when the predicate is multi-valued. It is necessary that the upper-bound of the synchronization sequence of the branches be a known (not f) sequence. A subtle point is determining the single-valued

[3] Our experience with the Split-C programs of Section 5 shows that this rule is nearly universally followed.

$$\frac{}{B,\ A \vdash i : +,\ A,\ \epsilon} \qquad \text{[Int]}$$

$$\frac{}{B,\ A \vdash id : A(id),\ A,\ \epsilon} \qquad \text{[Id]}$$

$$\frac{}{B,\ A \vdash communicate : -,\ A,\ \epsilon} \qquad \text{[Comm]}$$

$$\frac{}{B,\ A \vdash barrier : +,\ A,\ b} \qquad \text{[Barrier]}$$

$$\frac{}{B,\ A \vdash broadcast : +,\ A,\ r} \qquad \text{[Broadcast]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ \ldots \\ B,\ A_{n-1} \vdash Expr_n : a_n,\ A_n,\ s_n \\ B(f) = (a'_1, \ldots, a'_n), A \to a, A', s \\ A_n|dom(A) \preceq A \\ \forall 1 \leq i \leq n.\ a_i \preceq a'_i \end{array}}{B,\ A_0 \vdash f(Expr_1, \ldots, Expr_n) : a,\ A_n//dom(A') + A',\ s_1 \oplus \ldots \oplus s_n \oplus s} \qquad \text{[Fun]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ \ldots \\ B,\ A_{n-1} \vdash Expr_n : a_n,\ A_n,\ s_n \end{array}}{B,\ A_0 \vdash p\,(Expr_1, \ldots, Expr_n) : a_1 \sqcup \ldots \sqcup a_n,\ A_n,\ s_1 \oplus \ldots \oplus s_n} \qquad \text{[Prim]}$$

$$\frac{B,\ A \vdash Expr : a,\ A',\ s}{B,\ A \vdash x \leftarrow Expr : a,\ A'[x \leftarrow a],\ s} \qquad \text{[Assign]}$$

$$\frac{B,\ A[x \leftarrow +] \vdash Expr : a,\ A',\ s}{B,\ A \vdash let\ x\ in\ Expr : a,\ A'//\{x\},\ s} \qquad \text{[Let]}$$

$$\frac{\begin{array}{l} dom(A) = dom(A') = dom(A_0) \\ S = (a_1, \ldots, a_n), A \to a, A', s \\ A' = A''//\{x_1, \ldots, x_n\} \\ B[f \leftarrow S], A[x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n] \vdash Expr_1 : a,\ A'',\ s \\ B[f \leftarrow S], A_0 \vdash Expr_2 : a'_2,\ A_2,\ s_2 \end{array}}{B,\ A_0 \vdash letrec\ f(x_1, \ldots, x_n) = Expr_1\ in\ Expr_2 : a'_2,\ A_2,\ s_2} \qquad \text{[LetRec]}$$

$$\frac{\begin{array}{c} B,\ A_0 \vdash Expr_1 : +,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \\ B,\ A_1 \vdash Expr_3 : a_3,\ A_3,\ s_3 \end{array}}{B,\ A_0 \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : a_2 \sqcup a_3,\ A_2 \sqcup A_3,\ s_1 \oplus (s_2 \sqcup s_3)} \qquad \text{[If-Single]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : -,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \\ B,\ A_1 \vdash Expr_3 : a_3,\ A_3,\ s_3 \\ s_2 \sqcup s_3 \prec f \\ A' = A_1 \lhd (AV(Expr_2) \cup AV(Expr_3)) \end{array}}{B,\ A_0 \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : -,\ A',\ s_1 \oplus (s_2 \sqcup s_3)} \qquad \text{[If-Multi]}$$

$$\frac{\begin{array}{c} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \end{array}}{B,\ A_0 \vdash Expr_1;\ Expr_2 : a_2,\ A_2,\ s_1 \oplus s_2} \qquad \text{[Sequence]}$$

Figure 3: Inference rules.

variables of the final environment. Any variable that is modified in either branch could have different values in different processes on exit from the conditional; all of these variables must be marked multi-valued in the final environment. Let $AV(e)$ be the set of variables visible at e that may be assigned in the evaluation of e (including via function calls in e). A set $AV(e)$ is easily computed. Now define $A \lhd \{v_1, \ldots, v_n\}$ to be $A[v_1 \leftarrow -, \ldots, v_n \leftarrow -]$.

If the inference system of Figure 3 cannot assign any synchronization value to an expression, then evaluating the expression may cause processes to execute differing numbers of barriers and broadcasts—the program may get "out of synch". In this case the program is rejected. Of course, the inference system is conservative and may reject correct programs. We show in Section 5.1 that the system in fact works well on a suite of benchmarks.

3.1 Soundness

A sticky point in trying to prove our system correct is capturing the meaning of single-valued variables. Intuitively, a variable is single-valued if all processors have the same value for the variable at the same time. However, "at the same time" is a slippery notion in a setting with asynchronous execution. Only at points of global synchronization (i.e., `barriers`, `broadcasts`, and the start and end of execution) is it possible to assert anything useful about the state of all processes.

The key to this problem is to observe that the values of single-valued variables depend only on other single-valued expressions. Using this fact, it can be shown (without referring to time except within a single process) that if processes begin execution agreeing on single-valued inputs, then they terminate agreeing on the single-valued outputs.

The proof of soundness has two steps. First, we prove that single-valued outputs are determined solely by single-valued inputs for a process in isolation. Second, we show that if the inference rules can derive any proof for an expression, then all processes evaluating that expression execute the same sequence of synchronization operations.

A few definitions are required. Environments E_1 and E_2 are equal with respect to an abstract environment A, written $E_1 \approx_A E_2$, if $dom(E_1) = dom(E_2) = dom(A)$ and $\forall x.A(x) = + \Rightarrow E_1(x) = E_2(x)$. A function environment F and an abstract function environment B are *compatible*, written $F : B$, if $dom(F) = dom(B)$ and for all $f \in dom(F)$:

$$\begin{gathered} F(f) = f(x_1, \ldots, x_n) = Expr \\ B(f) = (a_1, \ldots, a_n), A \rightarrow a, A', s \\ B|F\,F(f), A[x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n] \vdash Expr : a, A'', s \\ A' = A''//\{x_1, \ldots, x_n\} \end{gathered}$$

An execution $\texttt{state}_1 \overset{*}{\underset{t}{\leadsto}} \texttt{state}_2$ is an execution with *synchronization sequence* t, where t is a string with one b for each `barrier` and one r for each `broadcast` executed. The *broadcast sequence* of an evaluation $[S_1, \ldots, S_n] \overset{*}{\leadsto} [S'_1, \ldots, S'_n]$ is the sequence of values returned by successive calls to `broadcast` during this evaluation.

Lemma 3.2 Let e be any expression and let $B, A \vdash e : a, A', s$. Let $E_1 \approx_A E_2$, and $F : B$. If

$$\begin{array}{ccc} [\langle F, E_1, C_1, e\rangle] & \overset{*}{\underset{t_1}{\leadsto}} & [C_1(E'_1, i_1)] \\ [\langle F, E_2, C_2, e\rangle] & \overset{*}{\underset{t_2}{\leadsto}} & [C_2(E'_2, i_2)] \end{array}$$

and the broadcast sequences of both evaluations are identical, then the following are all true:

- $t_1 = t_2$ and $t_2 \preceq s$
- $E'_1 \approx_{A'} E'_2$
- $a = + \Rightarrow i_1 = i_2$

Theorem 3.3 Let e be any expression and let $B, A \vdash e : a, A', s$. Let $F : B$ and $E_i \approx_A E_j$ for $i, j = 1..n$. Then

$$[\langle F, E_1, I, e\rangle, \ldots, \langle F, E_n, I, e\rangle] \overset{*}{\leadsto} [(E'_1, v_1), \ldots, (E'_n, v_n)]$$

or some process diverges.

The proofs of Lemma 3.2 and Theorem 3.3 are in Appendix A.

The semantics of Figure 2 does not handle synchronization errors, i.e. the cases where `barrier`s and `broadcast`s are mismatched, or when some processes waits at a `barrier` while other processes have terminated. In those cases, the evaluation hangs. Theorem 3.3 shows that this cannot occur with barrier inference: either the program terminates, or the evaluation sequence is infinite. Appendix D extends $\mathcal{L}$'s semantics with error checking rules for synchronization errors and then shows that Theorem 3.3 still holds, which proves that barrier inference makes runtime error checking for synchronization unnecessary.

3.2 Implementation

The only difficulty in translating the inference rules into an inference algorithm is in the determination of the assumptions to use in function environments. We define G, the *global abstract function environment* which is identical to the abstract function environment B, except that it contains the signatures of all functions of a program instead of those currently in scope. Using a global environment poses no problems as all function names are assumed to be unique.

A global function environment can be used to attempt to construct a proof:

$$\emptyset, \emptyset \vdash e : a, \emptyset, s$$

for an expression e by choosing $B = G|dom(B)$ at each step of the proof derived from the structure of e: the other quantities (A, A', a, s) can easily be computed once B is known. The proof thus constructed may however be incorrect. The goal of an implementation is to compute G such that a correct proof can be built from G, or to report that no proof exists, i.e. that program e is incorrect.

A value for G is found by recasting the inference rules as a function

$$\mathtt{I(G, A, Expr)} = \mathtt{(G', a, A', s, error)} : \mathtt{FEnv} \times \mathtt{AEnv} \times \mathtt{Expr} \rightarrow \mathtt{FEnv} \times \{+, -\} \times \mathtt{AEnv} \times \mathcal{S} \times \mathtt{Bool}$$

The definition of I is in Figure 4.

A call to $I(G, A, Expr)$ computes the properties of an expression assuming that all the functions behave as described in G and that A describes the single-valuedness of the free variables of $Expr$. The function I is total: when the [If-Multi] rule would fail, I simply returns an error indication in its last argument. This error is propagated back to the top-level expression. The [Fun] and [LetRec] inference rules express constraints on signatures in the abstract function environment: [Fun] requires that the single-valuedness of the arguments and free variables match the signature of the function, the [LetRec] rule requires that the inferred signature of the function's body equals the signature in the abstract function environment. When these constraints conflict with the function signatures of the global function environment G passed to I, I simply returns a new global function environment G', which satisfies all these constraints. Because the constraints depend on the assumed environment G, the computed environment G' may not satisfy the constraints that are computed by $I(G', A, Expr)$. However, a fixed point $G'' = I(G'', A, Expr)$ of I does satisfy all the constraints.

Theorem 3.4 Given a program p, all the following are true:

- The global abstract function environments G of p forms a lattice of finite height.
- I is monotonic in its G argument.

$$
\begin{array}{rcl}
I(G,A,i) & = & (G,+,A,\epsilon,false) \\
I(G,A,id) & = & (G,A(id),A,\epsilon,false) \\
I(G,A,communicate) & = & (G,-,A,\epsilon,false) \\
I(G,A,barrier) & = & (G,+,A,b,false) \\
I(G,A,broadcast) & = & (G,+,A,r,false) \\
I(G,A_0,f(Expr_1,\ldots,Expr_n)) & = & \text{let } (G_1,a_1,A_1,s_1,e_1) = I(G,A_0,Expr_1) \\
 & & \text{and } \ldots \\
 & & \text{and } (G_n,a_n,A_n,s_n,e_n) = I(G,A_{n-1},Expr_n) \\
 & & \text{and } G' = G_1 \sqcup \ldots \sqcup G_n \\
 & & \text{and } (a'_1,\ldots a'_n), A \rightarrow a, A', s = G(f) \\
 & & \text{and } (\overline{a'_1},\ldots \overline{a'_n}), \overline{A} \rightarrow \overline{a}, \overline{A'}, \overline{s} = G'(f) \\
 & & \text{in } (\ \ G'[f \leftarrow (\overline{a'_1} \sqcup a_1,\ldots,\overline{a'_n} \sqcup a_n), \overline{A} \sqcup (A_n|dom(A)) \rightarrow \overline{a}, \overline{A'}, \overline{s}], \\
 & & \qquad a, A_n//dom(A') + A', s_1 \oplus \ldots \oplus s_n \oplus s, e_1 \vee \ldots \vee e_n) \\
I(G,A_0,p\,(Expr_1,\ldots,Expr_n)) & = & \text{let } (G_1,a_1,A_1,s_1,e_1) = I(G,A_0,Expr_1) \\
 & & \text{and } \ldots \\
 & & \text{and } (G_n,a_n,A_n,s_n,e_n) = I(G,A_{n-1},Expr_n) \\
 & & \text{and } G' = G_1 \sqcup \ldots \sqcup G_n \\
 & & \text{in } (\ \ G', a_1 \sqcup \ldots \sqcup a_n, A_n, s_1 \oplus \ldots \oplus s_n, e_1 \vee \ldots \vee e_n) \\
 & & \\
I(G,A,x \leftarrow Expr) & = & \text{let } (G',a,A',s,e) = I(G,A,Expr) \text{ in } (G',a,A'[x \leftarrow a],s,e) \\
I(G,A,let\ x\ in\ Expr) & = & \text{let } (G',a,A',s,e) = I(G,A[x \leftarrow +],Expr) \text{ in } (G',a,A'//\{x\},s,e) \\
I(G,A_0,letrec\ f(x_1,\ldots,x_m) = Expr_1 & = & \text{let } (a_1,\ldots,a_m), A \rightarrow a, A', s = G(f) \\
in\ Expr_2) & & \text{and } (G_1,a'_1,A_1,s_1,e_1) = I(G,A[x_1 \leftarrow a_1,\ldots,x_m \leftarrow a_m],Expr_1) \\
 & & \text{and } (G_2,a'_2,A_2,s_2,e_2) = I(G,A_0,Expr_2) \\
 & & \text{and } G' = G_1 \sqcup G_2 \\
 & & \text{and } (\overline{a_1},\ldots,\overline{a_m}), \overline{A} \rightarrow \overline{a}, \overline{A'}, \overline{s} = G'(f) \\
 & & \text{in } (\ \ G'[f \leftarrow (\overline{a_1},\ldots,\overline{a_m}), \overline{A} \rightarrow a'_1, A_1//\{x_1,\ldots,x_m\}, s_1], \\
 & & \qquad a'_2, A_2, s_2, e_1 \vee e_2) \\
I(G,A_0,if\ Expr_1\ Expr_2\ else\ Expr_3) & = & \text{let } (G_1,a_1,A_1,s_1,e_1) = I(G,A_0,Expr_1) \\
 & & \text{and } (G_2,a_2,A_2,s_2,e_2) = I(G,A_1,Expr_2) \\
 & & \text{and } (G_3,a_3,A_3,s_3,e_3) = I(G,A_1,Expr_3) \\
 & & \text{and } G' = G_1 \sqcup G_2 \sqcup G_3 \\
 & & \text{and } e' = e_1 \vee e_2 \vee e_3 \\
 & & \text{and } A' = A_2 \sqcup A_3 \\
 & & \text{and } s' = s_1 \oplus (s_2 \sqcup s_3) \\
 & & \text{in } \ \text{if } e_1 = + \text{ then } (G', a_2 \sqcup a_3, A', s', e') \\
 & & \qquad \text{else } (G', -, A' \triangleleft (AV(Expr_2) \cup AV(Expr_3)), s', e' \vee (s_2 \sqcup s_3) = f) \\
I(G,A_0,Expr_1;Expr_2) & = & \text{let } (G_1,a_1,A_1,s_1,e_1) = I(G,A_0,Expr_1) \\
 & & \text{and } (G_2,a_2,A_2,s_2,e_2) = I(G,A_1,Expr_2) \\
 & & \text{in } (G_1 \sqcup G_2, a_2, A_2, s_1 \oplus s_2, e_1 \vee e_2)
\end{array}
$$

Figure 4: Inference Function

- A fixed point $I(G, \emptyset, p) = (G, a, A, s, false)$ exists iff a proof can be built with the inference rules of Figure 3.
- Any proof built from the inference rules defines a global abstract environment G' such that $G \preceq G'$, where G is the least fixed-point of I. Conversely, if there is any proof, then a proof can be constructed using G.

A proof is outlined in Appendix B.

The inference algorithm is thus:

1. Set G to the bottom element of the abstract function environment lattice.
2. Iterate $(G, a, A', s, error) = I(G, \emptyset, p)$ until G converges.
3. If $error$ is true, then report that p is erroneous.

In practice, we believe that a checking algorithm based on the language extensions of Section 4.1 is more important, and also easier to implement. We discuss our implementation of such a system in Section 5.1.

3.3 Examples

We conclude with example applications of the inference rules to Figures 1a and 1e. Other worked examples are included in Appendix C for the interested reader. The functions `random()` and `work()` do not contain barriers or modify visible variables.

Figure 1a fails the [If-Multi] rule - the alternatives of the `if` have different synchronization sequences.

$$\frac{\begin{array}{l}\emptyset,\emptyset \vdash \texttt{random()} : -,\ \emptyset,\ \epsilon \\ \emptyset,\emptyset \vdash \texttt{barrier} : +,\ \emptyset,\ b \\ \emptyset,\emptyset \vdash 0 : +,\ \emptyset,\ \epsilon \\ b \sqcup \epsilon = f \not\preceq f \text{ The rule fails}\end{array}}{\emptyset,\emptyset \vdash \texttt{if random() barrier else 0} :?} \quad \text{[If-Multi]}$$

Figure 1e successfully passes the inference rules, assuming `x` is single-valued:

$$\frac{\begin{array}{l}\emptyset,\{\texttt{x}:+\} \vdash \texttt{x} : +,\ \{\texttt{x}:+\},\ \epsilon \\ \emptyset,\{\texttt{x}:+\} \vdash \texttt{barrier} : +,\ \{\texttt{x}:+\},\ b \\ \emptyset,\{\texttt{x}:+\} \vdash \texttt{work()} : -,\ \{\texttt{x}:+\},\ \epsilon\end{array}}{\emptyset,\{\texttt{x}:+\} \vdash \texttt{if (x) barrier else work()} : -, \{\texttt{x}:+\}, \epsilon \oplus (b \sqcup \epsilon) = f} \quad \text{[If-Single]}$$

4 Realistic Languages

We now turn to the use of our techniques in realistic programming languages. Section 4.1 presents features we believe every SPMD language design should include. Section 4.2 applies barrier inference to heterogeneous parallel computing, while Section 4.3 discusses modifications needed to incorporate our techniques in programs written in C or FORTRAN-based languages.

4.1 SPMD Language Design

Current SPMD languages have few ways of indicating the synchronization structure of an application. Even with barrier inference, this makes SPMD programs unnecessarily difficult to read and maintain. We propose two language features that make this structure more explicit: named barriers and a `single` keyword to declare single-valued variables and functions.

Some SPMD languages provide *named barriers*, with the semantics that a runtime error results if processes simultaneously execute barriers with different names. Using named barriers indicates which syntactic `barrier`s may participate in a synchronization. Named barriers also make the difference between [If-Multi] and [If-Single] explicit: an [If-Multi] must use the same barrier names in both branches, while an [If-Single] should use different names. Usually named barriers are implemented using a broadcast (so the names can be compared) which is much slower than special-purpose barrier hardware (e.g., on the CM5 [17] and T3D [4]). But $\mathcal{L}$ already effectively has two barrier names: `barrier` and `broadcast`. Adding more names increases the alphabet of synchronization strings but has no impact on inference complexity. Our system thus allows named barriers to be checked at compile-time, allowing their implementation with the more efficient anonymous barriers. In a language with barrier inference there are only advantages to using named barriers.

Our inference system makes clear that knowing the single-valued variables is crucial to understanding an SPMD program's synchronization structure. We believe programmers ought to declare single-valued variables, formal parameters, and function results. These declarations are checked by a revised inference system. We propose a keyword `single` used as a type modifier (e.g., `single int x;`). The modifications to the language definition are:

$$\begin{array}{lcl}
\texttt{Expr} & ::= & \ldots \\
 & | & \texttt{let Decl in Expr} \\
 & | & \texttt{letrec Decl(Decl}, \ldots, \texttt{Decl}) = \texttt{Expr in Expr} \\
\\
\texttt{Decl} & ::= & \texttt{id} \\
 & | & \texttt{single id}
\end{array}$$

Declaring single-valuedness has two advantages. First, the program is clearer as the common parts of the data-flow are explicit. Second, barrier inference is simplified. Because abstract environments can be built directly from `single` declarations instead of computed, proofs

$$\texttt{B},\ \texttt{A} \vdash \texttt{Expr} : \texttt{a},\ \texttt{s}$$

no longer need a result environment. Function signatures

$$(\texttt{a}_1, \ldots, \texttt{a}_n) \rightarrow \texttt{a}, \texttt{s}$$

do not include environments either and can also be built from the declarations. Figure 5 shows the new inference rules.

4.2 Heterogeneous Computing

An interesting, though somewhat speculative, extension is to verify SPMD programs written for a heterogeneous environment, i.e. an environment that includes computers with different processor architectures and hence different data formats. A new problem arises: values that appear single-valued to the programmer may turn out to be slightly different at runtime because of differences between the architectures involved. For instance, they may be using slightly different precisions to compute intermediate results in floating point computations. Thus the innocuous loop

```
t = 0.0;
```

$$\frac{}{B,\ A \vdash i : +,\ \epsilon} \qquad \text{[Int]}$$

$$\frac{}{B,\ A \vdash id : A(id),\ \epsilon} \qquad \text{[Id]}$$

$$\frac{}{B,\ A \vdash communicate : -,\ \epsilon} \qquad \text{[Comm]}$$

$$\frac{}{B,\ A \vdash barrier : +,\ b} \qquad \text{[Barrier]}$$

$$\frac{}{B,\ A \vdash broadcast\ x : +,\ r} \qquad \text{[Broadcast]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : a_1,\ s_1 \\ \ldots \\ B,\ A \vdash Expr_n : a_n,\ s_n \\ B(f) = (a'_1, \ldots, a'_n) \rightarrow a, s \\ \forall 1 \leq i \leq n.\ a_i \preceq a'_i \end{array}}{B,\ A \vdash f(Expr_1, \ldots, Expr_n) : a,\ s_1 \oplus \ldots \oplus s_n \oplus s} \qquad \text{[Fun]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : a_1,\ s_1 \\ \ldots \\ B,\ A \vdash Expr_n : a_n,\ s_n \end{array}}{B,\ A \vdash p\,(Expr_1, \ldots, Expr_n) : a_1 \sqcup \ldots \sqcup a_n,\ s_1 \oplus \ldots \oplus s_n} \qquad \text{[Prim]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr : a,\ s \\ a \preceq A(x) \end{array}}{B,\ A \vdash x \leftarrow Expr : a,\ s} \qquad \text{[Assign]}$$

$$\frac{B,\ A[x \leftarrow a] \vdash Expr : a',\ s}{B,\ A \vdash let\ a\ x\ in\ Expr : a',\ s} \qquad \text{[Let]}$$

$$\frac{\begin{array}{l} S = (a_1, \ldots, a_m) \rightarrow a_0, s \\ B[f \leftarrow S], A[x_1 \leftarrow a_1, \ldots, x_m \leftarrow a_m] \vdash Expr_1 : a_0,\ s \\ B[f \leftarrow S], A \vdash Expr_2 : a'_2,\ s_2 \end{array}}{B,\ A \vdash letrec\ a_0\ f(a_1\ x_1, \ldots, a_m\ x_m) = Expr_1\ in\ Expr_2 : a'_2,\ s_2} \qquad \text{[LetRec]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : +,\ s_1 \\ B,\ A \vdash Expr_2 : a_2,\ s_2 \\ B,\ A \vdash Expr_3 : a_3,\ s_3 \end{array}}{B,\ A \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : a_2 \sqcup a_3,\ s_1 \oplus (s_2 \sqcup s_3)} \qquad \text{[If-Single]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : -,\ s_1 \\ B,\ A \vdash Expr_2 : a_2,\ s_2 \\ B,\ A \vdash Expr_3 : a_3,\ s_3 \\ s_2 \sqcup s_3 \prec f \\ \forall x. A(x) = + \Rightarrow x \notin (AV(Expr_2) \cup AV(Expr_3)) \end{array}}{B,\ A \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : -,\ s_1 \oplus (s_2 \sqcup s_3)} \qquad \text{[If-Multi]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : a_1,\ s_1 \\ B,\ A \vdash Expr_2 : a_2,\ s_2 \end{array}}{B,\ A \vdash Expr_1;\ Expr_2 : a_2,\ s_1 \oplus s_2} \qquad \text{[Sequence]}$$

Figure 5: Inference rules with a `single` keyword.

```
while (t < final_t) {
  t += d1 * d2;
  ...
  barrier();
}
```

might be executed a different number of times on different processors, even if they all have the same values for `d1`, `d2` and `final_t`.

Another problem in a heterogeneous environment is that different compilers are used to produce the executables. Thus any implementation-defined characteristics, such as order of evaluation in C, may vary. This could easily cause problems in code like:

```
a = f() + g();
```

where both `f` and `g` use barriers.

Our system can easily detect the former problem by supplying appropriate abstract signatures for primitive functions, reflecting whether those primitives are guaranteed to produce the same value in all processes. The second issue can be checked for by requiring that any set of statements whose order of evaluation is undefined have a synchronization sequence of ϵ, and that none of these statements modify any single-valued variables.

4.3 Application to Existing Languages

Some features of C and FORTRAN, which are popular starting points for SPMD languages, complicate barrier inference. Unstructured control-flow, aliasing, function pointers, and unitialised data structures are problematic. In this section we discuss how these language features can be handled. We have also extended these concepts to handle object-oriented programming and exception handling, but we do not report on this work here for lack of space.

4.3.1 Unstructured Control-Flow

Supporting unstructured control-flow (i.e., `goto`) requires the replacement of the [If-Single] and [If-Multi] rules by more complex mechanisms, though the inter-procedural aspects of the inference system remain unchanged. The problem can be divided into three parts: finding the single-valued variables, computing the synchronization sequence of a function, and verifying multi-valued branches do not cause synchronization problems.

The inference of single-valued variables is very similar to the problem of *binding-time analysis* in partial evaluation [12]: Given a set of variables whose value is assumed known (or single-valued in our case), determine which expressions and variables have a value that depends solely on these variables. The following algorithm is similar to [1], a binding-time analysis for C.

Finding single-valued variables

Outline: To find the single-valued variables of a function f:

1. Build the static, single-assignment (SSA) form [6] for function f. This has two advantages:
   (a) Each SSA variable is either single-valued or not. The status of variables at particular statements of the function is of no concern.
   (b) The points where different values of variables merge are explicit.

2. Build the *branch dependences* for each statement, i.e. the list of branch outcomes that determine whether a statement is executed. The branch dependences are computed from the control-dependence relation.

3. From the branch dependences associated with each assignment, determine for each ϕ-function in the SSA form which branches must be taken in a single-valued fashion for the value of the ϕ-function to be single-valued.

4. Determine for each SSA variable v its *dependence set*: All the variables that must be single-valued for v to be single-valued.

5. Build & solve a set of constraints whose solution gives the single-valued variables.

Appendix E details each step of this algorithm.

Computing the synchronization sequence is straightforward given a control-flow graph for a function: The abstract synchronization sequence from node n is defined to be the synchronization sequence executed from n to the function's exit-point. This sequence respects the control-flow equation:

$$\text{syncseq}(n) = \text{local-syncseq}(n) \oplus \bigsqcup_{s \in \text{succ}(n)} \text{syncseq}(s)$$

where local-syncseq(n) is the abstract synchronization sequence executed at node n. The value of syncseq(n) can be found by fixed-point iteration.

The final step is to verify that all the branches in the function are either single-valued (and correspond to [If-Single]) or that they obey the same restriction as the [If-Multi] rule, i.e. both paths have executed the same explicit synchronization sequence when they rejoin. The verification proceeds as follows for each multi-valued branch b of the control-flow graph:

- If b is branch-dependent on itself then it must form part of a loop. This loop cannot contain any synchronization statements, so local-syncseq(n) must be ϵ for all statements branch-dependent on b.
- Otherwise, b controls an if-like statement, and both paths must execute the same, known, synchronization statement. This is verified by computing the syncseq function defined above, restricted to b and all statements branch-dependent on b. The values of local-syncseq(n) for all other nodes n are temporarily considered to be $\perp$. If syncseq(b) $= f$ then branch b is invalid.

4.3.2 Other Language Features

The other language features mentioned above do not require such complex changes. In the presence of pointer values, detecting single-valued variables can require alias analysis, a well-known hard problem [15]. We have found that very conservative assumptions suffice in practice (see Section 5.1): a variable whose address is taken is multi-valued; any pointer dereference is multi-valued. Similar problems arise with function pointers, so we require that all functions whose address is computed have synchronization sequence ϵ, and we require that all visible variables they assign are multi-valued.

When a data structure is initialised with a single-valued expression at creation, it remains single-valued so long as all modifications are single-valued. Without initialization, detecting when all elements of a data structure are single-valued is much harder. Therefore we mark uninitialized data structures as multi-valued.

In practice we have found that pointers and complex data structures are rarely used in conjunction with synchronization. There are a few exceptions; in particular, in C programs command-line arguments are single-valued pointers and strings in `argv`. Many programs parse `argv` to initialize some single-valued variables. For these situations a mechanism is needed for the programmer to assert that a particular expression is single-valued. In the tradition of C, we call this a *single-valued cast*. Use of this feature should of course be minimized.

4.3.3 Single keyword in C

The `single` keyword proposed in Section 4.1 can be added to a C-based SPMD language. This keyword is a type qualifier, like `const` or `volatile`, that can be applied to any part of a type.

There is however one important restriction: if any component of a type t is declared `single`, then t must be implicitly considered to be `single` also. There are several reasons for this situation: first, a type such as "pointer to single int" is not useful, as the results of dereferencing it are not single-valued, and modifications made via such a pointer would violate the single-valuedness of the object pointed to. Secondly, a `struct` with a `single` field must obey the `single` restrictions when used as the destination of an assignment. The name equivalence used by C for `struct` types means that it is not possible to copy a structure with a single field to a structure that is identical except that that field is not `single`. Finally, it is not possible to copy arrays. Hence there are no useful types with a `single` component that are not themselves `single`.

We assume that all pointers are local and non-communicable: `single` is used to denote store whose computation is replicated across all processors, a remote pointer to `single` storage would allow this assumption to be violated. In a language that has remote pointers, the type referred to must not have any `single` components.

$\mathcal{L}$ has only integer variables, so all the copies of a `single` variable have equal values. When only some fields of a variable are single-valued it is inappropriate to talk of equality. Instead, we say that two variables are *consistent* if they agree on the values of those parts that are declared `single`. Formally, we say that two values of a type t are consistent if t is not `single` or:

- t is a base type and the values are equal (this is the only case addressed in $\mathcal{L}$).
- t is an array and all corresponding elements are consistent.
- t is a `struct` type and all `single` fields are consistent.
- t is an `union` type and the last assigned field is the same in both unions, and the values of that field are consistent.
- t is a function pointer and both values are null or point to the same function.
- t is a pointer and both values are null or refer to an object of the same size, these objects are consistent, and both pointers are at the same offset in this object.

The checking rules of Figure 5 extend naturally in this context. The $\preceq$ relation is replaced by the general rule that type `single` $t \preceq t$. Casts involving `single` are allowed, but are unchecked. Similarly, there can be no check that only the last assigned field of an `union` is read. Finally, all variables declared `single` must be initialised by single values, to guarantee that such variables are initially consistent across processes.

5 Experiments

We implemented a prototype of our inference system for Split-C [5], an explicitly parallel extension to C. We tested our prototype on Split-C kernels and applications. The empirical question we sought to answer is: How well does barrier inference integrate with real SPMD programming ? Our measure is the number of changes to preexisting programs required to conform to our system. The results were promising: the checks were all successful with minor changes, except for the exception handling aspects of one application. We also hand-examined the Splash-2 benchmarks and found that all but one would be checkable with our approach.

5.1 Split-C Prototype

For our purposes, the important features of Split-C are the `barrier()` and `all_bcast()` functions, which correspond to the `barrier` and `broadcast` primitives of $\mathcal{L}$.

The prototype is a cross between a pure inference system and the language extensions proposed in Section 4.1: It relies on a specification of the signatures of the functions and a list of the single-valued global variables, but infers the single-valued local variables. It verifies that all specifications are correct.

Our implementation follows the guidelines outlined in Section 4.3 for supporting C, except that we have not yet implemented the analysis of data structures (which was only needed by one of the Split-C programs). The algorithm for inferring single-valued variables is similar to [1].

Table 5.1 presents the programs and summarizes the results of the checking process. The second column counts the static occurrences of barriers in the program, while the third column reports the number of branches that controlled the execution of a barrier and whose condition was single-valued. The function signature and single-valued globals columns report the number of annotations that were necessary to check the program. The cases that required modifications to the code are summarized in the 'single-valued casts' and 'other changes' columns. Except for 'svd', all the casts are for values computed by parsing the program's arguments (see Section 4.3). The 'svd' algorithm uses single-valued arrays (not supported by our prototype), this accounts for 18 of the 19 casts. The last cast is due to a single-valued result being returned by reference, in C this implies taking the address of a variable: our system assumes that any variable whose address is taken is not single-valued.

The 'barnes' application includes exception handling (via `setjmp`), which is unchecked by our system[4]. This application also required one small, local change: It broadcasts values without using the Split-C `broadcast` primitives; we replaced this code with explicit broadcasts. One-line changes were needed in three programs, 'mm', 'wator' and 'nbody'. In these programs it was necessary to avoid taking the address of single-valued variables which were read with `scanf`. The second change in 'nbody' was to correct a minor bug detected by our prototype: when unexpected arguments were supplied only some processes exited.

These results show that our system is successful in verifying existing Split-C applications, with few changes and annotations. All but one of the programs depend on single-valued branches, which implies that conditional synchronization is the rule and not the exception in SPMD programs, and therefore that analysis of single-valued variables is necessary. The analysis time is low enough that our system can be integrated into an existing compiler without a large impact on execution time (the times, measured on an HP 715/80, represent the time spent in our system, they do not include the time to build the standard SSA representation used by our prototype).

5.2 The SPLASH-2 Benchmarks

As a further validation of our approach, we examined the synchronization structure of the SPLASH-2 benchmarks [25], which are written in C extended with macros for writing parallel programs. The facilities provided by the macros include named barrier synchronization. Process management is with a fork/join model, but all but one of the programs are effectively written in an SPMD style with all processes executing the same code (except for initialization). The exception is the 'radiosity' application; as it is outside our model we cannot check it.

Our implementation is written for Split-C and therefore does not check the SPLASH-2 programs. We examined the SPLASH-2 programs by hand to see if a suitably modified system would be able to check these programs. The results of this examination are given in Table 2. The four kernels and all but one of the applications pose no particular problems for our inference system.

[4] Checking use of `setjmp` and `longjmp` in C is almost impossible in any program analysis. In the 'barnes' application, when an exception arises in one process, the whole program is terminated.

Program	Lines	Number of barriers	Single-valued branches	Function signatures	Single-valued globals	Single-valued casts	Other changes	Analysis time
cannon	501	17	1	1	-	-	-	0.3s
cg	453	18	2	3	-	-	-	0.1s
cholesky	1542	38	16	4	-	2	-	2.3s
column	651	7	3	1	-	-	-	0.1s
fft3d	1181	12	5	1	-	1	-	0.1s
mm	508	23	1	1	-	-	1	0.2s
radix	379	7	3	-	-	2	-	0.1s
sample	302	9	0	-	-	-	-	0.1s
svd	1395	1	23	13	9	19 (or 1)[a]	-	0.2s
wator	348	10	5	-	3	-	2	0.1s
nbody	546	7	6	-	2	3	2	0.3s
em3d	1080	16	1	-	-	-	-	0.3s
barnes	2804	73	17	2	6	7	2	0.6s

[a] 18 of the 19 casts are required because of the lack of support of single-valued arrays.

Kernels:

- column, sample, radix: Sorting programs.
- cannon: Matrix multiplication using Cannons algorithm.
- cg: Solves a set of equations using the conjugent gradient method.
- cholesky: Seven different implementations of Cholesky decomposition.
- fft3d: A 3-dimensional fast fourier transform.
- mm: Matrix-multiply, blocked or unblocked.
- svd: Singular-value decomposition, using the Lanczos algorithm.

Applications:

- wator: Simulation of particle-like fish under current.
- nbody: A simple n body simulation code.
- em3d: 3-dimensional electro-magnetic simulation, described in [13].
- barnes: Simulate the interaction of a system of n bodies using the Barnes-Hut hierarchical method.

Table 1: Results of checking Split-C programs

6 Related Work

There are four strands of related work: SIMD (Single Instruction, Multiple Data) languages, synchronization analysis, binding-time analysis, and effect systems.

SIMD Languages divide variables into control unit and processing unit variables. Control unit variables resemble our single-valued variables: they are variables that have only one value. Unlike single-valued variables, control unit variables are stored in only one location. Control unit variables are declared with a `CU` keyword in the Illiac IV programming language Glypnir [16]. The Connection Machine language C* [23] calls these variables *scalar*. There is no equivalent of our inference system for these languages, as the properties we are inferring are guaranteed by SIMD semantics. Our proposed `single` keyword provides similar advantages for SPMD languages.

The ELP language [21] [24], a joint SIMD/SPMD programming language where both "modes" have the same semantics, allows declaration of single-valued variables with a `mono` keyword. When in SPMD mode the compiler guarantees that the single-valued property is preserved, presumably using rules similar to ours (the paper does not

Program	Lines	Number of barriers	Can be checked
ocean	2954	19	yes - needs single-valued array
	4703	20	inference (both versions)
barnes	2078	6	yes
fmm	3800	13	yes
radiosity	11319	5	no - not pure SPMD
raytrace	10020	1	yes
water	1744	9	yes
	2971	9	(both versions)
volrend	3704	13	yes
kernel cholesky	5050	4	yes
kernel fft	1005	7	yes
kernel lu	988	5	yes
	763	5	(both versions)
kernel radix	879	7	yes

Table 2: Results of examining the SPLASH-2 benchmarks

give many details on the checking strategy). ELP does not include explicit barriers or language-level broadcast, so there is no equivalent to our verification of synchronization. The programming model is also very different.

Analysis of the synchronization of parallel programs has been extensively studied for the purposes of deadlock and data-race detection as well as for optimisation. Our survey of this work is necessarily partial, and covers only static techniques.

Jeremiassen and Eggers [11] analyse barrier synchronization for SPMD programs to improve the precision of optimisation. They do not attempt to verify the correctness of the synchronization. Their analysis relies on named barriers for precision and does not consider single-valued variables, though they do consider dependencies on multi-valued constants like `pid` [10].

A number of papers analyse 2-way synchronization, such as post/wait or the accept/call mechanism of Ada, between explicitly specified tasks. As each task is specified with different code, there is no real analogue of single-valued variables. Analyzing synchronization in this context is similar to analyzing the synchronization between the two branches in the [If-Multi] case, for which we only allow very simple synchronization sequences. None of the following papers present exact solutions for more general situations.

One technique is to build a *concurrency graph* where nodes represent parallel program states, and edges represent synchronization or other state modifications. Taylor [22] considers only control-flow and the resulting graph can be exponential in the number of tasks. Young and Taylor [26] attempt to increase the precision of the concurrency graph by employing symbolic execution. Helmbold and McDowell [9] and McDowell [19] include data values in the concurrency states, and discuss a number of techniques for reducing the number of nodes.

A different approach is to determine which statements are executed before others, based on the synchronization statements. Callahan and Subhlok [2] and Callahan, Kennedy and Subhlok [3] compute an approximation of this relation and extend it with dependence distance information for loops. Masticola and Ryder [18] employ this information, along with other techniques, to compute a "can't happen together" relation for statements.

As mentioned in Section 4.3, inference of single-valued variables is similar to binding-time analysis [12]. The main difference is that we do not require that these values be directly computable from the initial set. Our single-valued variable inference algorithm is close to that presented by Auslander et al [1]. There is a difference in the handling of control-flow dependencies, and of course the purpose is unrelated.

Barrier Inference is an example of an *effect system* [7], where the effects are synchronization sequences and the type of a variable represents its single-valuedness.

7 Conclusion

We have identified an important property of SPMD programs that current languages do not explicitly support: The portion of control and data flow governing global synchronization is identical across all the processes. This synchronization kernel structures the entire application. We have developed an inference system that both detects this structure and verifies that global synchronization is correct. An implementation of this system for Split-C successfully checks a number of programs.

The synchronization kernel is sufficiently important that it should be explicitly visible in source code. We propose language features that make SPMD programs clearer and easier to check.

We are integrating these language extensions into a successor of Split-C based on Java [8], Titanium. This requires extending the application of the single-valued concept to more complex data structures, including references and objects, and to handle language features such as exception handling. We are also working on an algorithm that uses the results of our inference system to represent the portions of the code that may be executing simultaneously so that SPMD optimisations, e.g. [14], may be more precise.

References

[1] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 1996.

[2] D. Callahan and J. Subhlok. Static Analysis of Low-Level Synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, Madison, WI USA, [1] 1989. ACM Press , New York, NY , USA. Published as SIGPLAN Notices, volume 24, number 1.

[3] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of Event Synchronization in a Parallel Programming Tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, Seattle WA, March 1990.

[4] Cray Research Incorporated. *The CRAY T3D Hardware Reference Manual*, 1993.

[5] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. *Introduction to Split-C*. University of California, Berkeley, 1993.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 REFERENCE MANUAL. Technical Report MIT-LCS//MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[9] David P. Helmbold and Charles E. McDowell. Computing Reachable States of Parallel Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):76–84, December 1991.

[10] T. Jeremiassen and S. Eggers. Computing Per-Process Summary Side-Effect Information. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 175–191, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag.

[11] Tor E. Jeremiassen and Susan J. Eggers. Static Analysis of Barrier Synchronization in Explicitly Parallel Systems. In Michel Cosnard, Guang R. Gao, and Gabriel M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 171–180, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Co.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoff. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[13] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of the Supercomputing '93 Conference*, pages 262–273, Portland, OR, November 1993. IEEE Computer Society Press.

[14] Arvind Krishnamurthy and Katherine Yelick. Optimizing Parallel Programs with Explicit Synchronization. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 196–204, New York, NY, USA, June 1995. ACM Press.

[15] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[16] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal. Glypnir – A Programming Language for Illiac IV. *Communications of the ACM*, 18(3):157–164, March 1975.

[17] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. The Network Architecture of the CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, June 1992.

[18] S. P. Masticola and B. G. Ryder. Non-concurrency Analysis. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138, New York, NY, USA, July 1993. ACM Press.

[19] C. E. McDowell. A Practical Algorithm for Static Analysis of Parallel Programs. *Journal of Parallel and Distributed Computing*, 6(3):515–536, [6] 1989.

[20] Message Passing Interface Forum. Document for a standard message-passing interface. Technical report, University of Tennessee, Knoxville, Tenn., 1993.

[21] M. A. Nichols, H. J. Siegel, and H. G. Dietz. Data Management and Control-Flow Aspects of an SIMD/SPMD Parallel Language/Compiler. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):222–234, February 1993.

[22] Richard N. Taylor. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[23] Thinking Machines Corporation. *C* Programming Guide*, 1993.

[24] Lee Wang. *ELP User's Manual.* Parallel Processing Laboratory, School of Electrical and Computer Engineering, Purdue University, March 1996.

[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[26] M. Young and R. N. Taylor. Combining Static Concurrency Analysis with Symbolic Execution. In *Proceedings Workshop on Software Testing*, pages 10–18, 1986.

A Soundness

A.1 Proof of Lemma 3.2

The proof uses three simple lemmas. The first lemma states that all the continuations introduced by the rewrite rules are eventually applied. The second lemma asserts that if two evaluations have identical broadcast sequences, and that a prefix of both evaluations has the same the same synchronization sequence, then the remaining steps of both evaluations have the same broadcast sequence. Taken together, these two lemmas allow the rewrite rules to be broken into pieces so that an induction on the length of a rewrite sequence can be applied to all evaluations.

Lemma A.1 Let e be any expression. If $[\langle F, E, C, e\rangle] \underset{\epsilon}{\leadsto} [\langle F, E, C', e'\rangle] \overset{*}{\underset{t}{\leadsto}} [C(E', i)]$ then $[\langle F, E, C', e'\rangle] \overset{*}{\underset{t_1}{\leadsto}} [C'(E'', i')]$, $[C'(E'', i')] \overset{*}{\underset{t_2}{\leadsto}} [C(E', i)]$, and $t = t_1 \oplus t_2$.

Lemma A.2 If the broadcast sequences of

$$[\langle F, E_1, C_1, e_1\rangle] \overset{*}{\underset{t}{\leadsto}} [\langle F, E_1', C_1', e_1'\rangle] \overset{*}{\underset{t_1}{\leadsto}} [\langle F, E_1'', C_1'', e_1''\rangle]$$
$$[\langle F, E_2, C_2, e_2\rangle] \overset{*}{\underset{t}{\leadsto}} [\langle F, E_2', C_2', e_2'\rangle] \overset{*}{\underset{t_2}{\leadsto}} [\langle F, E_2'', C_2'', e_2''\rangle]$$

are identical, then the broadcast sequences of

$$[\langle F, E_1', C_1', e'\rangle] \overset{*}{\underset{t_1}{\leadsto}} [\langle F, E_1'', C_1'', e''\rangle]$$
$$[\langle F, E_2', C_2', e'\rangle] \overset{*}{\underset{t_2}{\leadsto}} [\langle F, E_2'', C_2'', e''\rangle]$$

are also identical.

The third lemma says that if e does not assign to x directly or via a function call, then x's value is unchanged by evaluation of e.

Lemma A.3 Let e be any expression, E any environment and F the free functions at e. If $[\langle F, E, C, e\rangle] \overset{*}{\leadsto} [C(E', i)]$ then $\forall x \in dom(E). x \notin AV(e) \Rightarrow E(x) = E'(x)$.

The proof of Lemma 3.2 proceeds by induction on the length of the rewrite sequence representing the evaluation of e. For each expression e, we can assume that $B, A \vdash e : a, A', s$, that $E_1 \approx_A E_2$, $F : B$, that

$$\begin{array}{lcl} [\langle F, E_1, C_1, e\rangle] & \overset{*}{\underset{t_1}{\leadsto}} & [C_1(E_1', i_1)] \\ [\langle F, E_2, C_2, e\rangle] & \overset{*}{\underset{t_2}{\leadsto}} & [C_2(E_2', i_2)] \end{array}$$

and that the broadcast sequences of both evaluations are identical. We must show that

- $t_1 = t_2$ and $t_2 \preceq s$
- $E_1' \approx_{A'} E_2'$
- $a = + \Rightarrow i_1 = i_2$

We consider each rewrite rule in turn.

- i: From the semantics and inference rules, we know

$$\frac{}{B,\ A \vdash i : +,\ A,\ \epsilon}$$
$$[\langle F, E_1, C_1, i\rangle] \underset{\epsilon}{\leadsto} [C_1(E_1, i)]$$
$$[\langle F, E_2, C_2, i\rangle] \underset{\epsilon}{\leadsto} [C_2(E_2, i)]$$

- ***id***: From the semantics and inference rules, we know

$$\frac{}{B,\ A \vdash id : A(id),\ A,\ \epsilon}$$
$$[\langle F, E_1, C_1, id\rangle] \underset{\epsilon}{\leadsto} [C(E_1, E_1(id))]$$
$$[\langle F, E_2, C_2, id\rangle] \underset{\epsilon}{\leadsto} [C(E_2, E_2(id))]$$

  So $A(id) = + \Rightarrow E_1(id) = E_2(id)$.

- ***communicate***: From the semantics and inference rules, we know

$$\frac{}{B,\ A \vdash communicate : -,\ A,\ \epsilon}$$
$$[\langle F, E_1, C_1, communicate\rangle] \underset{\epsilon}{\leadsto} [C_1(E_1, oracle())]$$
$$[\langle F, E_2, C_2, communicate\rangle] \underset{\epsilon}{\leadsto} [C_2(E_2, oracle())]$$

  The two different calls to $oracle()$ may return different values, but $a = -$.

- ***barrier***: From the semantics and inference rules, we know

$$\frac{}{B,\ A \vdash barrier : +,\ A,\ b}$$
$$[\langle F, E_1, C_1, communicate\rangle] \underset{b}{\leadsto} [C_1(E_1, 0)]$$
$$[\langle F, E_2, C_2, communicate\rangle] \underset{b}{\leadsto} [C_2(E_2, 0)]$$

- ***broadcast***: From the semantics, inference rules and the fact that the broadcast sequences of both evaluations are identical, we know

$$\frac{}{B,\ A \vdash broadcast : +,\ A,\ r}$$
$$[\langle F, E_1, C_1, broadcast\rangle] \underset{r}{\leadsto} [C_1(E_1, x)]$$
$$[\langle F, E_2, C_2, broadcast\rangle] \underset{r}{\leadsto} [C_2(E_2, x)]$$

  where $x = oracle()$.

- $\boldsymbol{x \leftarrow Expr}$: From the semantics, by hypothesis and Lemmas A.1 and A.2

$$[\langle F, E_1, C_1, x \leftarrow Expr\rangle] \underset{\epsilon}{\leadsto} [\langle F, E_1, C_1', Expr\rangle] \overset{*}{\underset{t_1}{\leadsto}} [C_1'(E_1', i_1)] = [C_1(E_1'[x \leftarrow i_1], i_1)]$$

$$[\langle F, E_2, C_2, x \leftarrow Expr\rangle] \underset{\epsilon}{\leadsto} [\langle F, E_2, C_2', Expr\rangle] \overset{*}{\underset{t_2}{\leadsto}} [C_2'(E_2', i_2)] = [C_2(E_2'[x \leftarrow i_2], i_2)]$$

$$\text{with } C_1' = \lambda(E', v).C_1(E'[x \leftarrow v], v) \text{ and } C_2' = \lambda(E', v).C_2(E'[x \leftarrow v], v)$$

The inference rule states

$$\frac{B,\ A \vdash Expr : a,\ A',\ s}{B,\ A \vdash x \leftarrow Expr : a,\ A'[x \leftarrow a],\ s}$$

By induction, we know that: $t_1 = t_2$, $t_2 \preceq s$, $E'_1 \approx_{A'} E'_2$ and $a = + \Rightarrow i_1 = i_2$. It follows that $E'_1[x \leftarrow i_1] \approx_{A'[x \leftarrow a]} E'_2[x \leftarrow i_2]$.

- ***let x in Expr***: From the semantics, by hypothesis and Lemmas A.1 and A.2

$$[\langle F, E_1, C_1, let\ x\ in\ Expr\rangle] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_1[x \leftarrow 0], C'_1, Expr\rangle] \overset{*}{\underset{t_1}{\rightsquigarrow}} [C'_1(E'_1, i_1)] = [C_1(E'_1//\{x\}, i_1)]$$

$$[\langle F, E_2, C_2, let\ x\ in\ Expr\rangle] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_2[x \leftarrow 0], C'_2, Expr\rangle] \overset{*}{\underset{t_2}{\rightsquigarrow}} [C'_2(E'_2, i_2)] = [C_2(E'_2//\{x\}, i_2)]$$

$$\text{with } C'_1 = \lambda(E', v).C_1(E'//\{x\}, v) \text{ and } C'_2 = \lambda(E', v).C_2(E'//\{x\}, v)$$

The inference rule states

$$\frac{B,\ A[x \leftarrow +] \vdash Expr : a,\ A',\ s}{B,\ A \vdash let\ x\ in\ Expr : a,\ A'//\{x\},\ s}$$

By induction, we know that: $t_1 = t_2$, $t_2 \preceq s$, $E'_1 \approx_{A'} E'_2$ and $a = + \Rightarrow i_1 = i_2$. It follows that $E'_1//\{x\} \approx_{A'//\{x\}} E'_2//\{x\}$.

- ***Expr*$_1$*; Expr*$_2$**: From the semantics, by hypothesis and Lemmas A.1 and A.2

$$\begin{gathered}
[\langle F, E_1, C_1, Expr_1;\ Expr_2\rangle] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_1, C'_1, Expr_1\rangle] \overset{*}{\underset{t_1^1}{\rightsquigarrow}} [C'_1(E'_1, i_1)] = [\langle F, E'_1, C_1, Expr_2\rangle] \overset{*}{\underset{t_2^1}{\rightsquigarrow}} [C_1(E''_1, j_1)] \\
[\langle F, E_2, C_2, Expr_1;\ Expr_2\rangle] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_2, C'_2, Expr_1\rangle] \overset{*}{\underset{t_1^2}{\rightsquigarrow}} [C'_2(E'_2, i_2)] = [\langle F, E'_2, C_2, Expr_2\rangle] \overset{*}{\underset{t_2^2}{\rightsquigarrow}} [C_2(E''_2, j_2)] \\
\text{with } C'_1 = \lambda(E', v).\langle F, E', C_1, Expr_2\rangle \text{ and } C'_2 = \lambda(E', v).\langle F, E', C_2, Expr_2\rangle
\end{gathered}$$

The inference rule states

$$\frac{\begin{matrix} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \end{matrix}}{B,\ A_0 \vdash Expr_1;\ Expr_2 : a_2,\ A_2,\ s_1 \oplus s_2}$$

Applying the induction hypothesis to $Expr_1$, we know that: $t_1^1 = t_1^2$, $t_1^2 \preceq s_1$, $E'_1 \approx_{A_1} E'_2$. The last fact completes the hypotheses of the induction for $Expr_2$ so we can also conclude that: $t_2^1 = t_2^2$, $t_2^2 \preceq s_2$, $E''_1 \approx_{A_2} E''_2$ and $a_2 = + \Rightarrow j_1 = j_2$. So $t_1^1 \oplus t_2^1 = t_1^2 \oplus t_2^2$ and $t_1^2 \oplus t_2^2 \preceq s_1 \oplus s_2$.

- ***f(Expr*$_1$*, ..., Expr*$_n$*)***: The arguments are evaluated in sequence, so the same inductive reasoning as for $Expr_1;\ Expr_2$ gives

$$\begin{gathered}
[\langle F, E_1, C_0^1, f(Expr_1, \ldots, Expr_n)\rangle] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_1, C_1^1, Expr_1\rangle] \overset{*}{\underset{t_1^1}{\rightsquigarrow}} [C_1^1(E_2^1, v_1^1)] \overset{*}{\underset{t_2^1}{\rightsquigarrow}} \ldots \overset{*}{\underset{t_n^1}{\rightsquigarrow}} [C_n^1(E_{n+1}^1, v_n^1)] \overset{*}{\underset{t_0^1}{\rightsquigarrow}} [C_0^1(E''_1, v_1)] \\
[\langle F, E_2, C_0^2, f(Expr_1, \ldots, Expr_n)\rangle] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_2, C_1^2, Expr_1\rangle] \overset{*}{\underset{t_1^2}{\rightsquigarrow}} [C_1^2(E_2^2, v_1^2)] \overset{*}{\underset{t_2^2}{\rightsquigarrow}} \ldots \overset{*}{\rightsquigarrow} t_n^2 [C_n^2(E_{n+1}^2, v_n^2)] \overset{*}{\underset{t_0^2}{\rightsquigarrow}} [C_0^2(E''_2, v_2)] \\
\text{with } F(f) = f(x_1, \ldots, x_n) = Expr \\
\text{and } C_1^1 = \lambda(E_2, v).\langle F, E_2, C_2^1, Expr_2\rangle, \ldots, C_n^1 = \lambda(E_{n+1}, v_n).\langle F|FF(f), E_0^1, C'_1, Expr\rangle \\
E_0^1 = (E_{n+1}|FV(f))[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] \\
C'_1 = \lambda E', v.C_0^1\ ((E_{n+1}//FV(f) + E'//\{x_1, \ldots, x_n\}), v) \\
\text{and } C_1^2 = \lambda(E_2, v).\langle F, E_2, C_2^2, Expr_2\rangle, \ldots, C_n^2 = \lambda(E_{n+1}, v_n).\langle F|FF(f), E_0^2, C'_2, Expr\rangle \\
E_0^2 = (E_{n+1}|FV(f))[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] \\
C'_2 = \lambda E', v.C_0^2\ ((E_{n+1}//FV(f) + E'//\{x_1, \ldots, x_n\}), v)
\end{gathered}$$

The inference rule states

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ \dots \\ B,\ A_{n-1} \vdash Expr_n : a_n,\ A_n,\ s_n \\ B(f) = (a'_1, \dots, a'_n), A \rightarrow a, A', s \\ A_n|dom(A) \preceq A \\ \forall 1 \leq i \leq n.\ a_i \preceq a'_i \end{array}}{B,\ A_0 \vdash f(Expr_1, \dots, Expr_n) : a,\ A_n//dom(A') + A',\ s_1 \oplus \dots \oplus s_n \oplus s}$$

Applying the induction hypothesis n times, we conclude: $t_i^1 = t_i^2$, $t_i^2 \preceq s_i$, $a_i = + \Rightarrow v_i^1 = v_i^2$, $E_{n+1}^1 \approx_{A_n} E_{n+1}^2$. We also know that $F : B$, i.e. $B|FF(f), A[x_1 \leftarrow a'_1, \dots, x_n \leftarrow a'_n] \vdash Expr : a, A'', s$ and $A' = A''//\{x_1, \dots, x_n\}$. As all function names in L are unique, $F|FF(f) : B|FF(f)$. From above

$$[C_n^1(E_{n+1}^1, v_n^1)] = [\langle F|FF(f), E_0^1, C'_1, Expr\rangle] \underset{t_0^1}{\overset{*}{\rightsquigarrow}} [C'_1(E'_1, v_1)] = [C_0^1(E''_1, v_1)]$$

$$[C_n^2(E_{n+1}^2, v_n^2)] = [\langle F|FF(f), E_0^2, C'_2, Expr\rangle] \underset{t_0^2}{\overset{*}{\rightsquigarrow}} [C'_2(E'_2, v_2)] = [C_0^2(E''_2, v_2)]$$

with $E''_1 = E_{n+1}^1//FV(f) + E'_1//\{x_1, \dots, x_n\}$ and $E''_2 = E_{n+1}^2//FV(f) + E'_2//\{x_1, \dots, x_n\}$

The hypothesis of the induction is thus verified, so $t_0^1 = t_0^2$, $t_0^2 \preceq s$, $a = + \Rightarrow v_1 = v_2$, $E'_1 \approx_{A''} E'_2$.

As

- $t_1^1 \oplus \dots t_n^1 \oplus t_0^1 = t_1^2 \oplus \dots t_n^2 \oplus t_0^2 \preceq s_1 \oplus \dots \oplus s_n \oplus s$
- $a = + \Rightarrow v_1 = v_2$
- By definition of the abstract function environment B, $FV(f) = dom(A) = dom(A')$. So, given that $E'_1 \approx_{A''} E'_2$, $E_{n+1}^1 \approx_{A_n} E_{n+1}^2$, it follows that $E''_1 \approx_{A_n//dom(A')+A'} E''_2$.

the lemma is verified for this case.

- ***p(Expr*$_1$*, ..., Expr*$_n$*)***: The arguments are evaluated in sequence, so the same inductive reasoning as above gives

$$[\langle F, E_1, C_0^1, p(Expr_1, \dots, Expr_n)] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_1, C_1^1, Expr_1\rangle] \underset{t_1^1}{\overset{*}{\rightsquigarrow}} \dots \underset{t_n^1}{\overset{*}{\rightsquigarrow}} [C_n^1(E_{n+1}^1, v_n^1)] = [C_0^1(E_{n+1}^1, p(v_1^1, \dots, v_n^1))]$$

$$[\langle F, E_2, C_0^2, p(Expr_1, \dots, Expr_n)] \underset{\epsilon}{\rightsquigarrow} [\langle F, E_2, C_1^2, Expr_1\rangle] \underset{t_1^2}{\overset{*}{\rightsquigarrow}} \dots \underset{t_n^2}{\overset{*}{\rightsquigarrow}} [C_n^2(E_{n+1}^2, v_n^2)] = [C_0^2(E_{n+1}^2, p(v_1^2, \dots, v_n^2))]$$

with $C_1^1 = \lambda(E_2, v).\langle F, E_2, C_2^1, Expr_2\rangle, \dots, C_n^1 = \lambda(E_{n+1}, v_n).C_0^1(E_{n+1}, p(v_1, \dots, v_n))$
and $C_1^2 = \lambda(E_2, v).\langle F, E_2, C_2^2, Expr_2\rangle, \dots, C_n^2 = \lambda(E_{n+1}, v_n).C_0^2(E_{n+1}, p(v_1, \dots, v_n))$

The inference rule states

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ \dots \\ B,\ A_{n-1} \vdash Expr_n : a_n,\ A_n,\ s_n \end{array}}{B,\ A_0 \vdash p(Expr_1, \dots, Expr_n) : a_1 \sqcup \dots \sqcup a_n,\ A_n,\ s_1 \oplus \dots \oplus s_n}$$

By induction, $t_i^1 = t_i^2$, $t_i^2 \preceq s_i$, $a_i = + \Rightarrow v_i^1 = v_i^2$, $E_{n+1}^1 \approx_{A_n} E_{n+1}^2$. As the value of p depends only on its arguments, $a_1 \sqcup \dots \sqcup a_n = + \Rightarrow p(v_1^1, \dots, v_n^1) = p(v_1^2, \dots, v_n^2)$.

- ***letrec f(x_1, ..., x_n) = Expr$_1$ in Expr$_2$***: From the semantics, by hypothesis and Lemmas A.1 and A.2

$$[\langle F, E_1, C_1, letrec\ f(x_1,\ldots,x_n) = Expr_1\ in\ Expr_2\rangle] \underset{\epsilon}{\leadsto} [\langle G, E_1, C_1, Expr_2\rangle] \overset{*}{\underset{t_1}{\leadsto}} [C_1(E_1', v_1)]$$
$$[\langle F, E_2, C_2, letrec\ f(x_1,\ldots,x_n) = Expr_1\ in\ Expr_2\rangle] \underset{\epsilon}{\leadsto} [\langle G, E_2, C_2, Expr_2\rangle] \overset{*}{\underset{t_2}{\leadsto}} [C_2(E_2', v_2)]$$
$$\text{with } G = F[f \leftarrow f(x_1,\ldots,x_n) = Expr_1]$$

The inference rule states

$$\frac{\begin{array}{l} dom(A) = dom(A') = dom(A_0) \\ S = (a_1,\ldots,a_m), A \rightarrow a, A', s \\ A' = A''//\{x_1,\ldots,x_n\} \\ B[f \leftarrow S], A[x_1 \leftarrow a_1,\ldots,x_n \leftarrow a_n] \vdash Expr_1 : a,\ A'',\ s \\ B[f \leftarrow S], A_0 \vdash Expr_2 : a_2',\ A_2,\ s_2 \end{array}}{B,\ A_0 \vdash letrec\ f(x_1,\ldots,x_n) = Expr_1\ in\ Expr_2 : a_2',\ A_2,\ s_2}$$

So $G : B[f \leftarrow S]$, therefore the induction hypothesis applies, and $t_1 = t_2$, $t_2 \preceq s_2$, $a_2' = + \Rightarrow v_1 = v_2$, $E_1' \approx_{A_2} E_2'$.

- ***if Expr$_1$ Expr$_2$ else Expr$_3$***: From the semantics, by hypothesis and Lemmas A.1 and A.2

$$[\langle F, E_1, C_1, if\ Expr_1\ Expr_2\ else\ Expr_3\rangle] \underset{\epsilon}{\leadsto} [\langle F, E, C_0^1, Expr_1\rangle] \overset{*}{\underset{t_1^1}{\leadsto}} [C_0^1(E_1', v_1)] \overset{*}{\underset{t_2^1}{\leadsto}} [C_1(E_1'', v_1')]$$
$$[\langle F, E_2, C_2, if\ Expr_1\ Expr_2\ else\ Expr_3\rangle] \underset{\epsilon}{\leadsto} [\langle F, E, C_0^2, Expr_1\rangle] \overset{*}{\underset{t_1^2}{\leadsto}} [C_0^2(E_2', v_2)] \overset{*}{\underset{t_2^2}{\leadsto}} [C_2(E_2'', v_2')]$$
$$\text{with } C_0^1 = \lambda(E', v).\langle F, E', C_1,\ \text{if } v = 0 \text{ then } Expr_2 \text{ else } Expr_3\rangle$$
$$\text{and } C_0^2 = \lambda(E', v).\langle F, E', C_2,\ \text{if } v = 0 \text{ then } Expr_2 \text{ else } Expr_3\rangle$$

The inference rule applied to this construction is either [If-Single] or [If-Multi]. If the rule is [If-Single]

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : +,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \\ B,\ A_1 \vdash Expr_3 : a_3,\ A_3,\ s_3 \end{array}}{B,\ A_0 \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : a_2 \sqcup a_3,\ A_2 \sqcup A_3,\ s_1 \oplus (s_2 \sqcup s_3)}$$

By induction, $v_1 = v_2$, $t_1^1 = t_1^2 \preceq s_1$ and $E_1' \approx_{A_1} E_2'$, so the applications of C_0^1 and C_0^2 return states that evaluate the same expression. If $v_1 = v_2 = 0$, we get

$$[[C_0^1(E_1', 0)] = [\langle F, E_1', C_1, Expr_2\rangle] \overset{*}{\underset{t_2^1}{\leadsto}} [C_1(E_1'', v_1')]$$
$$[C_0^2(E_2', 0)] = [\langle F, E_2', C_2, Expr_2\rangle] \overset{*}{\underset{t_2^2}{\leadsto}} [C_2(E_2'', v_2')]$$

The hypothesis of the induction is satisfied, so $t_2^1 = t_2^2 \preceq s_2$, $E_1'' \approx_{A_2} E_2''$ and $a_2 = + \Rightarrow v_1' = v_2'$.
The case for $v_1 = v_2 \neq 0$ is similar. It therefore follows that $t_1^1 \oplus t_2^1 = t_1^2 \oplus t_2^2 \preceq s_1 \oplus (s_2 \sqcup s_3)$, $a_2 \sqcup a_3 = + \Rightarrow v_1' = v_2'$ and $E_1'' \approx_{A_2 \sqcup A_3} E_2''$, so the lemma is verified for this case.
If the inference rule is [If-Multi]

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : -,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \\ B,\ A_1 \vdash Expr_3 : a_3,\ A_3,\ s_3 \\ s_2 \sqcup s_3 \prec f \\ A' = A_1 \triangleleft (AV(Expr_2) \cup AV(Expr_3)) \end{array}}{B,\ A_0 \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : -,\ A',\ s_1 \oplus (s_2 \sqcup s_3)}$$

If v_1 and v_2 are both equal to or different from 0, then the [If-Multi] behaves like the [If-Single] case, and the lemma is easily verified as the $\triangleleft$ operator only weakens the requirements.

Assuming, with no loss of generality, that $v_1 = 0$ and $v_2 \neq 0$ the lemma is applied independently to each expression

$$\begin{aligned}[C_0^1(E_1', v_1)] &= [\langle F, E_1', C_1, Expr_2\rangle] \overset{*}{\underset{t_2^1}{\rightsquigarrow}} [C_1(E_1'', v_1')] \\ [C_0^2(E_2', v_2)] &= [\langle F, E_1', C_1, Expr_3\rangle] \overset{*}{\underset{t_2^2}{\rightsquigarrow}} [C_2(E_2'', v_2')]\end{aligned}$$

By induction, we get: $t_2^1 \preceq s_2$ and $t_2^2 \preceq s_3$. Also t_2^1 and t_2^2 are strings in $\{b, r\}^*$, so $\bot \prec t_2^1 \preceq s_2$, $\bot \prec t_2^2 \preceq s_3$, and $s_2 \sqcup s_3 \prec f \Rightarrow t_2^1 = s_2 = s_3 = t_2^2 = s_2 \sqcup s_3$.

We have $E_1' \approx_{A_1} E_2'$. By Lemma A.3

$$\begin{aligned} &\forall x \notin AV(Expr_2).E_1'(x) = E_1''(x) \\ &\forall x \notin AV(Expr_3).E_2'(x) = E_2''(x) \\ \Rightarrow &\forall x \notin (AV(Expr_2) \cup AV(Expr_3)).E_1'(x) = E_2'(x) \Rightarrow E_1''(x) = E_2''(x) \\ \Rightarrow &E_1'' \approx_{A_1 \triangleleft (AV(Expr_2) \cup AV(Expr_3))} E_2'' \end{aligned}$$

so the lemma is verified for this case.

A.2 Proof of Theorem 3.3

The following simple lemma asserts that if an evaluation terminates, then the special continuation I must have been evaluated:

Lemma A.4 Let e be any expression, E any environment and F any function environment. If

$$[\langle F, E, I, e\rangle] \overset{*}{\underset{t}{\rightsquigarrow}} [(E', v)]$$

then

$$[\langle F, E, I, e\rangle] \overset{*}{\underset{t}{\rightsquigarrow}} [I(E', v)]$$

To prove Theorem 3.3 we must show that given an expression e, a proof $B, A \vdash e : a, A', s$, and environments $F, E_1, \ldots, E_n$ such that $F : B$ and $E_i \approx_A E_j$ for $i, j = 1..n$, that:

$$[\langle F, E_1, I, e\rangle, \ldots, \langle F, E_n, I, e\rangle] \overset{*}{\rightsquigarrow} [(E_1', v_1), \ldots, (E_n', v_n)]$$

or some process diverges.

The proof is simple. Lemma A.4 and the assumption that no process diverges implies that for all i

$$[\langle F, E_i, I, e\rangle] \overset{*}{\underset{t_i}{\rightsquigarrow}} [I(E_i', v_i)] = [(E_i', v_i)]$$

We assume, with no loss of generality, that the sequence of values returned by *broadcast* is the same for all evaluations. It then follows from Lemma 3.2 that $t_1 = t_2 = \ldots = t_n$.

The i evaluation sequences can therefore be combined into one common evaluation sequence as they all have the same synchronization sequence: denoting the k'th element of t_i by t_i^k, each individual evaluation sequence can be decomposed as (t_i^k used as an expression stands for the synchronization operation corresponding to the synchronization letter)

$$[\langle F, E_i, I, e\rangle] \overset{*}{\underset{\epsilon}{\rightsquigarrow}} [\langle F, E_i^1, C_i^1, t_i^1\rangle] \underset{t_i^1}{\rightsquigarrow} [C_i^1(E_i^1, v_i^1)] \overset{*}{\underset{\epsilon}{\rightsquigarrow}} [\langle F, E_i^2, C_i^2, t_i^2\rangle] \underset{t_i^2}{\rightsquigarrow} \cdots \underset{t_i^m}{\rightsquigarrow} [\langle C_i^m(E_i^m, v_i^m)\rangle] \overset{*}{\underset{\epsilon}{\rightsquigarrow}} [(E_i', v_i)]$$

As $t_i^k = t_j^k$ for all i, j, k these individual evaluations can be combined into a global evaluation using the general interleaving rule for individual processes and the *barrier* and *broadcast* rules as follows

$$[\langle F, E_1, I, e\rangle, \ldots, \langle F, E_n, I, e\rangle] \overset{*}{\underset{\epsilon}{\leadsto}} [\langle F, E_1^1, C_1^1, t_1^1\rangle, \ldots, \langle F, E_n^1, C_n^1, t_1^1\rangle] \underset{t_1^1}{\leadsto} \cdots$$
$$\cdots \underset{t_1^m}{\leadsto} [C_1^m(E_1^m, v_1^m), \ldots, C_n^m(E_n^m, v_n^m)] \overset{*}{\underset{\epsilon}{\leadsto}} [(E_1', v_1), \ldots, (E_n', v_n)]$$

This completes the proof.

B Implementation Soundness

Theorem 3.4 has four parts

1. The global abstract function environments G of p forms a lattice of finite height.

   Proof: There are only a finite number of functions in a program and each component of a function signature is a lattice of finite height.

2. I is monotonic in its G argument.

   Proof: We prove that all the results of I are monotonic in both G and A. The proof is a straightforward induction on the structure of expressions. In particular, $\oplus$ is monotonic.

3. A fixed point $I(G, \emptyset, p) = (G, a, A, s, false)$ exists iff a proof can be built with the inference rules of Figure 3.

   Proof: Given a proof for a program p, an environment G is built from all the assumptions about signatures embodied in applications of the [LetRec] rule. It is easy to verify that $I(G, \emptyset, p) = (G, a, A, s, false)$ for such a G. The only case that can set $error$ to $true$ is an *if*, which occurs only if $s_2 \sqcup s_3 = f$, which is precluded by the existence of a proof.

   To prove the converse, we consider a slightly less restrictive version of the inference rules of Figure 3: we remove the $s_2 \sqcup s_3 \prec f$ requirement from [If-Multi]. It is obvious that all proofs in the old system are still valid in the new one.

   Given a fixed point G of I, $I(G, \emptyset, p) = (G, a, A, s, error)$ it is easy to build a proof in this expanded inference system: The requirements of the [Fun] rule are implied by G being a fixed point, the assumptions needed for [LetRec] are read from G. There is thus a one-to-one correspondence between proofs in the expanded systems and fixed points of I. All fixed points for which $error = true$ are valid in the new system, but not in the old, while those for which $error = false$ are valid in both. If all fixed points have $error = true$, it will not be possible to build a proof in the old system. Thus a fixed point with $error = false$ exists iff a proof exists in the old system.

4. Any proof built from the inference rules defines a global abstract environment G' such that $G \preceq G'$, where G is the least fixed-point of I. A proof can be built from G if any proof exists.

   Proof: From point 3 it follows that the environment G' defined by any proof is a fixed point $I(G', \emptyset, p) = (G', a', A', s', false)$. The least fixed point G of I satisfies the equation $I(G, \emptyset, p) = (G, a, A, error)$. By definition, $G \preceq G'$. From point 2, we conclude that $error \preceq false$, i.e. $error = false$. So a proof can be built from G.

C Examples

This appendix shows the results produced by our inference system on the more complex examples from Figure 1. The while loops of Figures 1b and 1e are rewritten using `letrec` so that we can directly apply the rules in Figure 3. Figure 6 shows the new code.

Example (b)	Example (f)

<pre>letrec w1() = if random()
                (barrier; w1())
              else
                0
  in w1();
work1(); barrier();
work2(); barrier();
work3();</pre>
	<pre>i <- 0;
	letrec w2() = if (i < 10)
	                (if (i = 1) barrier;
	                 i <- i + 1;
	                 w2())
	              else
	                0
	  in w2();
	barrier;</pre>

Figure 6: Loops rewritten with `letrec`

- Figure 1b fails [If-Multi]. We end up trying to match

$$\frac{\begin{array}{l}\{\mathtt{w1} : (), \emptyset \rightarrow +, \emptyset, \bot\}, \emptyset \vdash \mathtt{random()} : -,\ \emptyset,\ \epsilon \\ \{\mathtt{w1} : (), \emptyset \rightarrow +, \emptyset, \bot\}, \emptyset \vdash \mathtt{(barrier;\ w1())} : +,\ \emptyset,\ b \\ \{\mathtt{w1} : (), \emptyset \rightarrow +, \emptyset, \bot\}, \emptyset \vdash 0 : +,\ \emptyset,\ \epsilon \\ b \sqcup \epsilon = f \not\prec f \text{ The rule fails}\end{array}}{\vdash \mathtt{if\ (random())\ (barrier;\ w1())\ else\ 0} :?} \quad \text{[If-Multi]}$$

- Figure 1f succeeds with this signature for `w2`: $(), (\mathtt{i} : +) \rightarrow +, (\mathtt{i} : +), f$.

- Figure 1g successfully passes [If-Multi]

$$\frac{\begin{array}{l}\vdash \mathtt{random()} : -,\ \emptyset,\ \epsilon \\ \vdash \mathtt{(barrier;\ barrier)} : +,\ \emptyset,\ bb \\ \vdash \mathtt{(work1();\ barrier;\ work2();\ barrier)} : +,\ \emptyset,\ bb \\ bb \sqcup bb = bb \prec f\end{array}}{\vdash \mathtt{if\ (random())\ (...)\ \ else\ (...)} : -, \emptyset, bb} \quad \text{[If-Multi]}$$

- Figure 1h fails because both branches have abstract synchronization sequence f

$$\frac{\begin{array}{l}\vdash \mathtt{random()} : -,\ \emptyset,\ \epsilon \\ \vdash \mathtt{(while\ ...)} : +,\ \emptyset,\ f \\ \vdash \mathtt{(j\ =\ i\ +\ 10;\ \ ...)} : +,\ \emptyset,\ f \\ f \sqcup f = f \not\prec f \text{ The rule fails}\end{array}}{\vdash \mathtt{if\ (random())\ (...)\ \ else\ (...)} :?} \quad \text{[If-Multi]}$$

D Runtime Error Checking

Figure 7 adds new semantics rules to $\mathcal{L}$ that detect the following runtime errors: mismatch of `barrier` and `broadcast`, and termination of some processes while others are waiting at a `barrier` or `broadcast`.

$$C \quad \texttt{Cont} = \texttt{Env} \times \mathcal{N} \rightarrow \texttt{State}$$
$$\texttt{State} = \texttt{FunEnv} \times \texttt{Env} \times \texttt{Cont} \times \texttt{Expression} + \texttt{Env} \times \mathcal{N} + \perp$$

$$[\ldots, \langle F_i, E_i,\ C_i,\ broadcast\rangle, \ldots, \langle F_j, E_j,\ C_j,\ barrier\rangle, \ldots] \ \rightsquigarrow\ [\perp, \ldots, \perp]$$

$$[\ldots, (E_i, v_i), \ldots, \langle F_j, E_j,\ C_j,\ barrier/broadcast\rangle, \ldots] \ \rightsquigarrow\ [\perp, \ldots, \perp]$$

Figure 7: Semantic rules for runtime synchronization error detection

Theorem 3.3 is now stronger, as it implies that an evaluation does not terminate as $[\perp, \ldots, \perp]$. As it is impossible to apply the new semantic rules of Figure 7 to a single process, Lemma 3.2 is valid in the new system, and therefore so is Theorem 3.3.

As a consequence, barrier inference guarantees that a program cannot have a mismatch of a `barrier` or `broadcast` and also that processes cannot wait at a `barrier` or `broadcast` when some processes of the SPMD program have terminated. This eliminates the need for runtime error checking of these conditions.

E Unstructured Single Inference

This appendix gives additional details and examples on the inference of single-valued variables in unstructured control-flow graphs. Global variables are considered as implicit arguments and results of functions and are otherwise treated exactly as local variables.

Branch Outcome Dependences

A statement s is *directly branch dependent* on outcome o of branch b if $s \in$ CD(b) and s postdominates o, where CD(b) is the set of statements control-dependent on b, and an *outcome* of a branch is one of its successors.

The `branch-dependences` relation is the closure of the direct branch dependence relation.

Figure 8 shows the branch dependences for three statements in a simple control-flow graph. Statement `s2` is interesting because it depends on both outcomes of condition `a`. This captures the intuition that the outcome of decision `a` is important to whether statement `s2` gets evaluated, in that it determines what other condition (`b` or `c`) gets tested to directly determine whether `s2` gets executed or not. All of `a, b, c` must be single-valued for all processes in a Split-C program to get the same value of `x`.

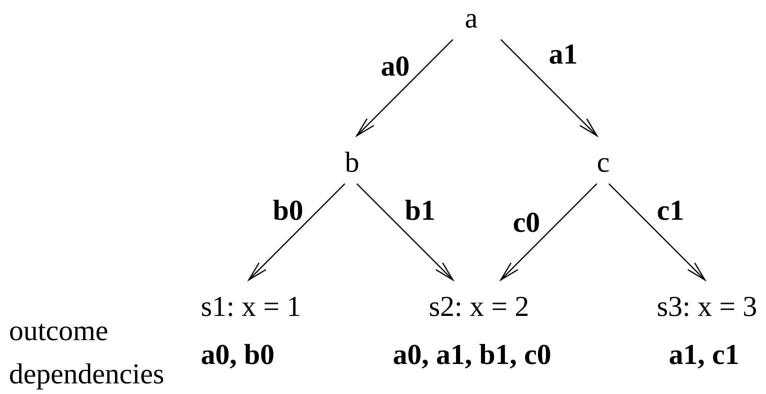


Figure 8: Branch outcome dependences

Single-valuedness at ϕ-functions

The value of a ϕ-function $\phi(v_1, v_2)$ depends on branch b iff different outcomes for branch b are found in

`branch-dependences(definition(`v_1`))`

and

`branch-dependences(definition(`v_2`))`

where `definition(`v`)` is the statement where v is assigned. The set of branches on which ϕ-function s depends is called ϕ`-dependences(`s`)`.

Any ϕ-function with more than 2 arguments is handled by considering all pairs of variables.

Figure 9 adds some control-flow merges to Figure 8. The branch dependences are:

- `x4` depends on the `a` and `b` branches as `x1` and `x2` have different branch outcomes in their branch dependence sets. Notice that `x4` is not dependent on outcomes of branch `b`.
- `x5` depends on the `a` and `c` branches. It doesn't depend on `b` directly, but it depends on `x4` that depends on `b`.

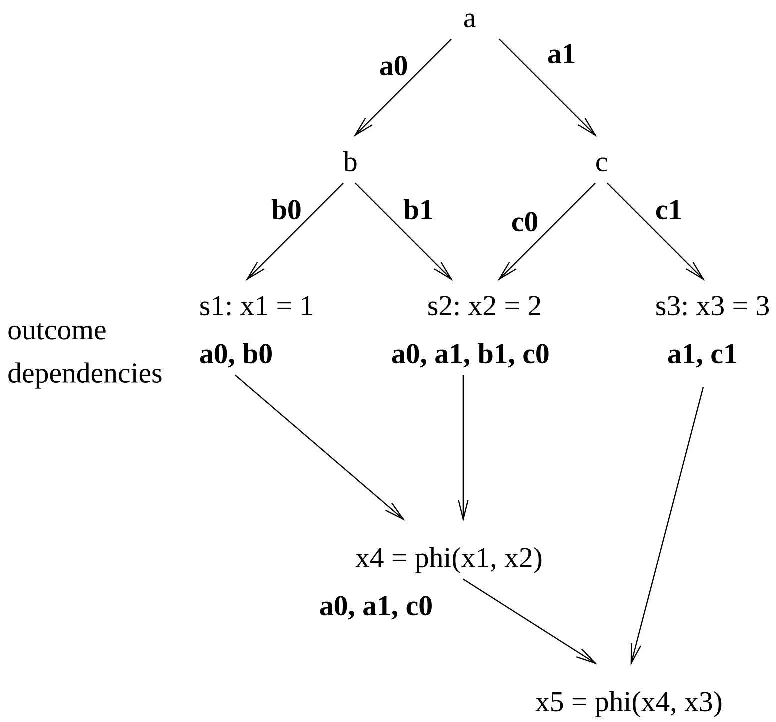


Figure 9: Branch dependences at ϕ-functions

Dependence Sets

The dependence set for v is the set of variables that must be invariant for v to be invariant. There are three cases:

1. v is the result of an assignment $v = op(v_1, v_2, \ldots)$. $\texttt{var-dependences}(v) = \{v_1, v_2, \ldots\}$.
2. v is the result of an assignment $v = \phi(v_1, v_2, \ldots)$.

$$\texttt{var-dependences}(v) = \{v_1, v_2, \ldots\} \cup (\bigcup_{b \in \phi\texttt{-dependences}(v=\phi(v_1,v_2,\ldots))} \texttt{branch-variables}(b))$$

   where $\texttt{branch-variables}(s)$ is the set of variables that determine the outcome of branch statement s.

3. v is assigned in some other fashion (e.g. a function call). v has no `var-dependences` set.

Building the Constraints

The maximal solution of this set of constraints gives the set of single-valued variables. Variables are either known to be single-valued, known not to be single-valued, or depend on other variables.

For every variable v that has a dependence set, add the constraint

$$(\bigwedge_{w \in \texttt{var-dependences}(v)} w) \Leftrightarrow v$$

A $false \Leftrightarrow v$ is added for every input argument that is not single-valued, and for every function call result that is not single-valued (global variables are considered implicit arguments to and results of functions).

The language semantics may mandate the addition of other $false \Leftrightarrow v$ constraints, e.g. pointer dereferences.

Solving the Constraints

The following algorithm finds a maximal solution of the set of constraints S over variables V:

```
truevars = V
while S contains a constraint 'false <=> v'
  truevars = truevars - { v }
  S = S - { 'false <=> v' }
  replace all constraints 'w1 & ... & wn <=> w' in S whose left hand side
    contains v with 'false <=> w'
end
```

`truevars` is the maximal solution.