Analyzing Stores and References in a Parallel Symbolic Language

Suresh Jagannathan* Stephen Weeks‡

Abstract

We describe an analysis of a parallel language in which processes communicate via first-class mutable shared locations. The sequential core of the language defines a higher-order strict functional language with list data structures. The parallel extensions permit processes and shared locations to be dynamically created; synchronization among processes occurs exclusively via shared locations.

The analysis is defined by an abstract interpretation on this language. The interpretation is efficient and useful, facilitating a number of important optimizations related to synchronization, processor/thread mapping, and storage management.

1 Introduction

Sequential programming languages have long been the target of sophisticated compile-time optimization and analysis techniques. In particular, there has been much success in applying optimizations derived via static analysis to expressive symbolic programming languages such as Scheme [7], Prolog [11], Self [4], or ML [1].

There has also been much work in building concurrent analogues of such languages (*e.g.,* MultiLisp [15], CML [28], ML Threads [25], Concurrent Smalltalk [16], Concurrent Prolog [29], etc.). There has been relatively little effort, however, in applying semantic analysis techniques suitably modified to handle concurrency to explicitly parallel symbolic programming languages.

The introduction of concurrency complicates compile-time analysis because programs may now define multiple threads of control that may be introduced dynamically during program execution. Defining a practical and useful analysis that tracks inter- and intra-thread control-flow for such programs requires a framework more sophisticated in approach and structure than one required to handle sequential programs.

In this paper, we address the problem of deriving store and reference information from parallel programs written in a higher-order programming language. We do so by developing a novel abstract interpretation [10] of this language. The language we present supports dynamic lightweight process creation in which processes communicate via shared locations; synchronization occurs via these locations. Its sequential core defines a higher-order call-by-value functional language with list data structures. The dynamic semantics of this core is thus similar to pure Scheme or ML; the parallel extensions provide functionality similar to that found in asynchronous parallel dialects of Lisp [13, 15, 19], ML [8, 25], and other higher-order parallel languages [2].

The interpretation computes inter- and intra-thread control and data-flow information for this language; in particular, the interpretation collects information on where shared locations are created, mutated, and referenced. The information associated with a location L includes (a) L's creation point, (b) the program points at which L is referenced parameterized on a per-thread basis, and (c) the program points at which L is written parameterized on a per-thread basis.

Our abstract interpretation is efficient. There is a polynomial-time algorithm, and a complete implementation written in Scheme; the implementation has proven to be efficient in practice. The output of the interpretation is also useful. Because the analysis tracks the creation, reference, and mutation points of shared data, it is possible to infer an approximation to the dynamic communication patterns of a program. This information can be used to implement a number of important optimizations concerned with thread scheduling, thread mapping, and data locality.

The abstract interpretation is described via an operational semantics. The exact semantics is given by a transition relation on program states, where a program state describes all currently executing threads and the locations that have been created. The approximate semantics is defined by a transition function on approximate program states which associate an approximate environment with every combination of program label (or continuation) and thread creation point. Hence, the complexity of the interpretation is con-

*Computer Science Division, NEC Research Institute, Princeton, NJ. suresh@research.nj.nec.com.
‡Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA. sweeks@cs.cmu.edu.

To appear in the 1994 Proceedings of the ACM Conference on Lisp and Functional Programming, June 1994.

trolled by selectively collapsing ground values, closures, and potential interleavings among threads; these approximations are performed on a per-thread basis.

The paper is organized as follows. The next section presents related work. Section 3 describes the kernel language used. Section 4 provides motivation for the problem by presenting a simple example, and the information derived by the analysis on this example; we also present benchmark results obtained by using the analysis to guide synchronization, storage allocation, and thread scheduling strategies. Section 5 presents the operational semantics for the language. Section 6 defines an approximation to this semantics. Correctness proofs are given in Section 7, and Section 8 presents conclusions.

2 Related Work

Chow and Harrison [6] present an abstract interpretation for a parallel language with **cobegin-coend** statements. Although the motivation for their work is similar to ours, there are numerous differences in the technical development. First, our kernel language permits arbitrary process creation in which synchronization is mediated only via shared variable access. Their language uses a more restrictive **cobegin-coend** form that constrains all processes created at a **cobegin** to synchronize at a **coend** barrier point before continuing. Their language also does not support list structures. Second, the efficiency of their interpretation is exponential in the size of the input program, significantly weakening the practicality of their results. Third, their analysis is erroneous in its treatment of programs in which **cobegin** branches have multiple instantiations of concurrently executing threads [5]; this seriously limits its utility.

Mercouroff [24] presents an abstract interpretation of a CSP-style language. He assumes no shared variables (and thus no global effect changes), and no higher-order procedures or processes. There has also been work in data flow analysis and deadlock detection for concurrent systems that communicate via message passing [27]. Such systems do not consider operations on shared variables, nor are they applicable to programs with higher-order procedures that may initiate such operations.

The formal underpinnings of our interpretation technique are similar to those underlying formal optimization frameworks devised for functional [3, 17] and logic programming [11] languages. Insofar as we use abstract interpretation [10] as an optimization tool for higher-order languages, our work also bears close resemblance to type recovery and flow analysis algorithms developed for languages such as Scheme [18, 30, 31], and to alias and lifetime analysis for related higher-order languages [12]. The presence of concurrency separates our formulation from these in some significant respects. For example, our analysis maintains environments and continuations on a per-thread basis and does not completely collapse closures of the same function or data structures created in distinct threads.

There has also been much work on *dynamic* monitoring of operations on shared locations in parallel programs, *e.g.*, [23, 32]. The goal in these efforts is to guarantee deterministic behavior of parallel programs using runtime detection techniques. Programs that contain data races which may lead to indeterminate results are considered erroneous; history and access information is maintained at runtime to determine when a data race on a shared location occurs. Our concern is not to prohibit indeterminacy in parallel programs, but to capture useful control and dataflow information statically that can be used to improve runtime performance. Furthermore, unlike [23, 32] which only consider programs that use fork-join style parallelism, our analysis makes no assumption on the structure of the process graph generated.

FX [22] and Jade [21] are two languages that permit users to add annotations to a serial program which are then used by a compiler to generate parallel code. In both FX and Jade, programmers can declare side-effect information; in addition, a Jade program may be partitioned into pieces that can be executed concurrently without side-effect conflicts. In contrast, we assume an explicitly parallel language that is amenable to static analysis; the results of this analysis can be used either by a programmer, compiler, or runtime system to generate more efficient parallel code.

3 The Kernel Language

Our language (see Fig. 1) has constants, variables, functions, primitive applications, call-by-value function applications, conditionals, recursive function definitions, and a process creation operation. The primitives include operations for creating and accessing pairs and shared locations. Constants include integers and booleans.

Shared locations in this language provide a natural communication medium through which concurrently evaluating threads may sensibly transmit data. Shared locations are created using the primop **mk-loc**, and are initially unbound. If x is a location, both **read**(x) and **remove**(x) return the value of x, blocking if x is unbound. In addition, if x is bound, **remove**(x) marks x as unbound. Blocked **remove** and **read** expresions may unblock when a **write** subsequently occurs on the location of interest. The act of writing or reading a location, or removing a location's contents is atomic.

To create a lightweight thread of control to evaluate e, we evaluate **spawn** e. **Spawn** returns immediately after creating a new thread; its value is TRUE. The environment in which a newly spawned thread evaluates is the same as its parent thread. Threads are completely asynchronous, and communicate exclusively via shared locations.

For example, the MultiLisp [15] expression, (**future** e) , is equivalent to,

let *loc* = (**mk-loc**)
in begin
  spawn write(loc, e)
  loc
end

and (**touch** v) is equivalent to,

$$
\begin{array}{lcl}
e & \in & \mathit{Exp} \\
c & \in & \mathit{Const} = \mathit{Int} + \mathit{Bool} \\
x & \in & \mathit{Var} \\
f & \in & \mathit{Func} \\
p & \in & \mathit{Prim} = \{\mathbf{cons}, \mathbf{car}, \mathbf{cdr}, \textbf{mk-loc}, \mathbf{write}, \mathbf{read}, \mathbf{remove}, \ldots\} \\
\\
e & ::= & c \\
 & | & x \\
 & | & f \\
 & | & p(e_1 \ldots\ e_n) \mid (p) \\
 & | & e_1\,(e_2\ e_3\ \ldots\ e_n) \mid (e) \\
 & | & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
 & | & \mathbf{letrec}\ x_1 = f_1\ \ldots\ x_n = f_n\ \mathbf{in}\ e \\
 & | & \mathbf{spawn}\ e \\
\\
f & ::= & \lambda x_1\ x_2\ \ldots\ x_n . e
\end{array}
$$

Figure 1: The kernel language.

$$
\begin{array}{l}
\mathbf{if\ location?}\ (v) \\
\quad \mathbf{then\ read}(v) \\
\quad \mathbf{else}\ v
\end{array}
$$

(Note that "**let**" and "**begin**" are syntactic sugar for simple and nested application, respectively.)

As another example, the following expression implements a simple *fetch-and-op* abstraction [14] using locations. The procedure representing this abstraction atomically returns the current contents of the cell, and stores a new value using the procedure argument provided.

$$
\begin{array}{l}
\lambda\, \mathit{init}.\, \mathbf{let}\ \mathit{cell}\ =\ (\textbf{mk-loc}) \\
\quad \mathbf{in\ begin} \\
\qquad \mathbf{write}(\mathit{cell}, \mathit{init}) \\
\qquad \lambda\, \mathit{op}.\, \mathbf{let}\ \mathit{val}\ =\ \mathbf{remove}(\mathit{cell}) \\
\qquad\quad \mathbf{in\ begin} \\
\qquad\qquad \mathbf{write}(\mathit{cell}, \mathit{op}\,(\mathit{val})) \\
\qquad\qquad \mathit{val} \\
\qquad\quad \mathbf{end} \\
\quad \mathbf{end}
\end{array}
$$

Labels

For the purposes of the exact as well as the approximate semantics defined in the following sections, each expression in a program is labeled with entry and exit labels. Let $l, l' \in$ *Label*, and let e be an expression. Then $[\,e\,]^l_{l'}$ means that l (e's entry label) is the program point immediately before execution of e and l' (e's exit label) is the program point immediately after execution of e. Exit labels will also be used to hold temporary values during the execution of a program. For instance, l' in the expression $[\,e\,]^l_{l'}$ will hold the value to which e evaluates. (In the following, we omit entry and exit labels whenever appropriate.)

In both the exact and approximate semantics, *spawn labels* are treated specially. In expression $[\,\mathbf{spawn}\,[\,e\,]^s_{l''}\,]^l_{l'}$, s is a spawn label as well as e's entry label. Labels l, l', and l'' are not spawn labels. Spawn labels merely syntactically label process creation points. We denote the set of spawn labels by $\mathit{Slabel} \subset \mathit{Label}$.

4 An Example

Fig. 2 gives an example of a program written in the language described above. This program generates a tree of processes; children communicate results to their parent via shared locations *left* and *right*; these locations represent edges in the tree. Parents communicate values to their children via shared location *my-val* ; this location gets bound to *parent-val* in recursive calls to this procedure.

Consider the intra- and inter-thread control-flow of this program. Threads generated at **spawn** points s_1 and s_2 communicate only through locations written by their parent (via locations bound to *parent-val*) or their children (via locations bound to *left* and *right*). Thus, not all locations created are referenced or written by all threads, and not all threads communicate with one another. Since the control and dataflow properties of this program are not obvious by trivial examination of the source text, an analysis that computes this information would be beneficial.

Our interpretation maps each pair of program label and spawn label to an approximate environment that maps variables and labels to sets of approximate values. For a particular pair $\langle l, s\rangle$, the approximate environment represents the combination of every environment that might exist in a dynamic instantiation of a thread created at spawn point s which is currently executing at program label l. In the example shown, there are three spawn expressions: an implicit top-level **spawn**, which we denote by the reserved spawn label s_0, and two **spawn** expressions internal to procedure f with spawn labels s_1 and s_2. Fig. 3 shows relevant bindings in these environments. The approximate value of a location is a pair of labels, $\langle l, s\rangle$ indicating that the location was created at program label l by a thread instantiated at spawn label s.

```
letrec f = λ (edge parent-val op)
             let my-val = [ (mk-loc) ]^l0
                 left =   [ (mk-loc) ]^l1
                 right =  [ (mk-loc) ]^l2
             in if (g)
                  then begin
                         [ spawn f(left my-val left-part) ]^s1
                         [ spawn f(right my-val right-part) ]^s2
                         [ write(my-val, (g1)) ]^l3
                         let v = op ([ read(left) ]^l4
                                     [ read(right) ]^l5
                                     [ read(parent-val) ]^l6 )
                         in [ write(edge, v) ]^l7
                       end
                  else [ write(edge, (g2)) ]^l8
in let init = [ (mk-loc) ]^l9
   in begin
        [ write(init, (g3)) ]^l10
        f([ (mk-loc) ]^l11 init root)
      end
```

Figure 2: A simple fine-grained tree-structured parallel program. A left child computes its result using function *left-part*; a right child computes its result using *right-part*.

For example, consider threads created by the **spawn** expression at s_1. The approximate value of *parent-val* at program label l_6 is a set containing three approximate locations; the first corresponds to locations allocated by the **mk-loc** operation at label l_0 when evaluated by the top-level thread, the second and third correspond to the locations allocated by the same **mk-loc** operation evaluated within threads created at s_1 and s_2, respectively.

The information collected reveals the following:

1. Threads created by **spawn** expressions at spawn point s_1 write l_0 locations (*i.e.,* locations bound to *my-val*) created by s_1 threads; threads created by **spawn** expressions at spawn point s_2 write l_0 locations created by s_2 threads.
2. Threads created by **spawn** expressions at spawn point s_1 read *left* and *right* locations created by s_1; threads created by **spawn** expressions at spawn point s_2 read *left* and *right* locations created by s_2.
3. Threads created by **spawn** expressions at spawn point s_1 and s_2 read l_0 locations (*i.e.,* locations bound to *my-val*) created by s_0, s_1, and s_2 threads.
4. Threads created by **spawn** expressions at spawn label s_1 write *left* locations created in s_0, s_1, and s_2 threads, but never write *right* locations; threads created by **spawn** expressions at spawn label s_2 write *right* locations created in s_0, s_1, and s_2 threads, but never write *left* locations.
5. Locations are only read and written in this program; a location once written never has its contents removed.

In the following, we consider an *abstract location* to be the set of location instances associated with a given (**mk-loc**) program point created during program evaluation. An *abstract thread* defines the set of thread instances associated with a given **spawn** label instantiated during program evaluation.

Two heuristics that may lead to significantly improved performance can be applied based on this information:

1. Allocate threads with the same spawn label on the same processor if they read and write the same abstract location L. Such a mapping may lead to improved memory and communication locality since references and stores of L instances do not involve any inter-processor communication.
2. If an abstract location L is created, written, and read only by an abstract thread T, allocate a separate heap for all instances of L created by T. Partitioning such locations may lead to more efficient storage management strategies since only instances of T need to be involved in any garbage collection of L's heap. Moreover, allocating a separate heap L_H for L may lead to improved locality and communication benefits since allocations of other locations made by T (or other abstract threads) are not interleaved with allocations to L_H.

In addition, because the contents of a location is never removed once written, writers never need to synchronize with blocked readers when writing a value into a non-empty location. Since readers executing concurrently with such writers

Program Labels	Spawn Labels		
	s_0	s_1	s_2
l_3	$\mathit{my\text{-}val} \mapsto \{\langle l_0, s_0\rangle\}$	$\mathit{my\text{-}val} \mapsto \{\langle l_0, s_1\rangle\}$	$\mathit{my\text{-}val} \mapsto \{\langle l_0, s_2\rangle\}$
l_4	$\mathit{left} \mapsto \{\langle l_1, s_0\rangle\}$	$\mathit{left} \mapsto \{\langle l_1, s_1\rangle\}$	$\mathit{left} \mapsto \{\langle l_1, s_2\rangle\}$
l_5	$\mathit{right} \mapsto \{\langle l_2, s_0\rangle\}$	$\mathit{right} \mapsto \{\langle l_2, s_1\rangle\}$	$\mathit{right} \mapsto \{\langle l_2, s_2\rangle\}$
l_6	$\mathit{parent\text{-}val} \mapsto \{\langle l_9, s_0\rangle\}$	$\mathit{parent\text{-}val} \mapsto \{\langle l_0, s_0\rangle, \langle l_0, s_1\rangle, \langle l_0, s_2\rangle\}$	$\mathit{parent\text{-}val} \mapsto \{\langle l_0, s_0\rangle, \langle l_0, s_1\rangle, \langle l_0, s_2\rangle\}$
l_7, l_8	$\mathit{edge} \mapsto \{\langle l_{11}, s_0\rangle\}$	$\mathit{edge} \mapsto \{\langle l_1, s_0\rangle, \langle l_1, s_1\rangle, \langle l_1, s_2\rangle\}$	$\mathit{edge} \mapsto \{\langle l_2, s_0\rangle, \langle l_2, s_1\rangle, \langle l_2, s_2\rangle\}$

Figure 3: Salient binding information derived by the abstract interpretation for the program shown in Fig. 2.

are guaranteed to see either the new or old contents of a location, readers only need to acquire a lock if the location being read is empty; a reader can never block on a location that has already been written by some other thread.

To optimize the example shown in Fig. 2, a thread created at spawn label s_1 maps its s_1-created child onto its processor; a thread created at spawn label s_2 maps its s_2-created child onto its processor. Furthermore, *left* locations created by threads instantiated at spawn label s_1 are allocated on a separate heap, as are *right* locations created by threads instantiated at spawn label s_2.

To test the utility of the analysis and these optimizations, we integrated the output generated by the interpretation on a version of the tree program[1], with an implementation of the above heuristics and optimization. The transformed program compiles to Sting [20], a multi-threaded dialect of Scheme. The 8 processor wallclock times obtained for the optimized and unoptimized versions are shown in Fig. 4. (The unoptimized implementation allocates all locations on a single shared heap, uses a simple round-robin thread scheduling policy, and acquires a lock on every write and read to a shared location.) The times indicate roughly a factor of 10 speedup over a range of tree depths in the fully optimized case, and a factor of close to 4 applying just the heuristics.

Depth	*Locations*	*Unoptimized*	*Optimized*	
5	O(32K)	3.14	.82	.31
6	O(64K)	6.19	1.70	.67
7	O(128K)	13.33	3.36	1.31
8	O(256K)	24.98	6.71	2.61

Figure 4: Optimized vs. unoptimized times for a variant of the program shown in Fig. 2. Times were recorded on an 8 processor 150MHz Silicon Graphics MIPS R4400 shared-memory multiprocessor. The first time in the optimized column measures the effect of applying the mapping and heap allocation heuristics. The second time, in addition, also incorporates the effect of eliminating lock acquistions and releases for writes and reads of non-empty locations. All times are in seconds.

5 Exact Semantics

We develop the abstract interpretation by first defining an operational semantics for the language, and then defining an abstract interpretation [10] of that semantics.

The exact semantics of a program $P \in \mathit{Exp}$ is defined by a transition system $\langle \mathit{State}, \rightarrow_\tau\rangle$ specific to P, where *State* is the set of configurations and $\rightarrow_\tau \subseteq \mathit{State} \times \mathit{State}$ is a single-step transition relation. We define a transition function $F \in \mathbf{Pow}(\mathit{State}) \rightarrow \mathbf{Pow}(\mathit{State})$ as

$$F(\Phi) = \{\phi \mid \exists \phi' \in \Phi \,.\, \phi' \rightarrow_\tau \phi\}.$$

The strongest global invariant of program P is the set of all configurations reachable during the execution of P. Define $\mathit{lfp}(g, x)$ to be the least fixpoint of g above x, *i.e.,* the least fixpoint of $\lambda y.\, x \sqcup g(y)$. The strongest global invariant of P is expressible as a fixpoint $\mathit{lfp}(F, \Phi_0)$, where Φ_0 is the set of initial configurations of P.

5.1 Configurations

A configuration has two components. The first component is the state of all executing threads, and the second component is the global store, or heap.

In a single configuration, many threads may be executing. A typical representation for a configuration in an operational semantics might be a set of thread states, each element in the set containing a program counter (or the thread's current continuation), process id, and environment. For the purposes of the approximate semantics to follow, however, we depart slightly from this standard intuitive notion. Our representation of the executing threads in a program is a *thread map.* Given a pair of labels $\langle l, s\rangle$, the thread map returns a set of thread states corresponding to those threads whose spawn point is s and whose program counter is l. A *spawn point* of an executing thread is the spawn label at which the thread was created. This partitioning of the set of executing threads by program label and spawn point will be slightly awkward in the exact semantics, but will prove quite useful when we define the approximate semantics.

For instance, consider the expression

[1] The program implemented allocates a 1000 element vector of locations at each leaf; a random number is stored in each location. An internal node in the tree sends a combining procedure to its two children via the location bound to *my-val*; this procedure takes two vectors of locations and returns the vector containing the smallest sum. An internal node applies this procedure to the vectors returned by its children.

$$
\begin{array}{rcrcl}
\phi & \in & State & = & ThreadMap \times Heap \\
\upsilon & \in & ThreadMap & = & (Label \times Slabel \rightarrow Threads) \\
\pi & \in & Threads & = & (Env \times Kont)^{*} \\
\rho & \in & Env & = & Var + Label \rightarrow D \\
\kappa & \in & Kont & = & (Label \times Env)^{*} \\
\delta & \in & Heap & = & Locs \times Pairs \times Closures \\
\psi & \in & Locs & = & Ptr \rightarrow D + \{\text{UNBOUND}\} \\
\omega & \in & Pairs & = & Ptr \rightarrow D \times D \\
\theta & \in & Closures & = & Ptr \rightarrow Env \\
d & \in & D & = & Const + Ptr \\
\sigma & \in & Ptr & = & Label \times Slabel \times Nat
\end{array}
$$

Figure 5: A configuration (state) in the exact semantics.

$$
\begin{array}{l}
[\textbf{spawn if } [\, e_1 \,]^{l} \\
\quad\quad \textbf{then } [\, \textbf{spawn } [\, e_2 \,]^{l_2} \,]^{s_2} \\
\quad\quad \textbf{else } \; [\, \textbf{spawn } [\, e_3 \,]^{l_3} \,]^{s_3}]^{s_1}
\end{array}
$$

Let t be a thread spawned at label s_1. Because t's spawn point is s_1, when t is at the entry point of e_1, the thread map maps $\langle l, s_1 \rangle$ to a set which has t's state as an element. Another thread is subsequently spawned, either for e_2 or e_3; the spawn point during execution of e_2 or e_3 would be s_2 or s_3, respectively. By definition, the (implicitly spawned) top-level thread has spawn label s_0. Note that there may be multiple threads created from a spawn point that are simultaneously evaluating.

A single thread's state is an ⟨*environment*, *continuation* ⟩ pair, where environments map program variables and exit labels to values. A continuation is a list of ⟨*label*, *environment* ⟩ pairs. Each element in this list represents a stack frame in the dynamic call chain of the thread. *Label* corresponds to the return address of the frame, and *environment* corresponds to the frame's display.

The second component of a configuration is the heap that is shared by all threads. It is partitioned into regions for shared locations, pairs, and closures. A pointer into the heap is a triple $\langle l, s, i \rangle$, where l and s are the program label and spawn point, respectively, at which the allocation of the corresponding data object occurred, and i is an integer used to disambiguate allocations made from different dynamic occurrences of the same program label and spawn point.

5.2 The Transition Rules

The single-step transition relation $\rightarrow_{\tau}$ is shown in Fig. 6. It is defined in terms of an auxiliary relation $\rightsquigarrow_s$ (Fig. 7), parameterized by a spawn-point label s, that expresses the "sequential" component of the semantics. The relation is given in two parts; the first describes the transition step for all expressions other than **spawn**; the second describes the transition step for **spawn**. The latter entails establishing the base continuation for the new thread in the thread map.

Because of the blocking semantics for shared variable operations, each primitive operation p is defined in terms of a relation $\hookrightarrow_p$; salient elements of this relation are shown in Fig. 8.

6 Approximate Semantics

We would like to achieve a computable invariant of program $P \in Exp$, but as the strongest global invariant described in Section 5 is in general not computable, we must describe a weaker invariant. We do this via an approximate semantics based on abstract interpretation.

The approximate semantics of a program $P \in Exp$ is defined by a transition system $\langle \widehat{State}, \widehat{F} \rangle$ specific to P, where $\widehat{State}$ approximates $\mathbf{Pow}(State)$ and $\widehat{F}$ approximates F. The approximate (and computable) global invariant of P will be an element of $\widehat{State}$ that approximates the strongest global invariant $lfp(F, \Phi_0)$, where Φ_0 is the set of initial configurations of P. This will be done by finding an appropriate fixpoint of $\widehat{F}$.

6.1 Configurations

An approximate configuration is a finite approximation of a set Φ of exact configurations. The domain of approximate configurations is defined in Fig. 9. An approximate configuration has two parts:

- The first part is an approximate thread map: a function that maps a pair $\langle l, s \rangle$ to an environment that approximates all environments of any thread in any configuration in Φ that is at program label l and spawn point s. The structure chosen for an exact configuration is convenient since it partitions threads in a configuration into $\langle l, s \rangle$ pairs. Note that this approximation means that there is no disambiguation between multiple instances of a thread created at the same spawn point executing at the same program label. Also note that the approximate thread state does not include its dynamic context.
- The second part of an approximate configuration is the heap. Approximate pointers do not disambiguate multiple allocations made from the same program label and spawn point. As a result, there is no need to allocate closures in the heap; the environment of a closure pointer $\langle l, s \rangle$ is simply the environment to which the thread map maps $\langle l, s \rangle$.

Since we are not concerned with constants, they are abstracted to the singleton set.

Each equation in Fig. 9 defines the set of elements in a (complete) lattice. The lattice $\langle \widehat{State}, \bot, \top, \sqcup, \sqcap \rangle$ is constructed pointwise and elementwise from the lattice $\langle \widehat{D}, \emptyset, \{\text{CONST}\} + Label \times Slabel, \cup, \cap \rangle$ with the exception that the lattice of $\widehat{Env}$ is the lift of the lattice of $Var + Label \rightarrow \widehat{D}$ with $\bot_{\widehat{\text{Env}}} = \text{ABSENT}$.

$$\frac{\begin{array}{c} \upsilon\langle l,s\rangle = \langle t_1,\ldots,t_i,\langle\rho,\kappa\rangle,t_{i+1},\ldots,t_m\rangle \qquad \upsilon\langle l',s\rangle = \langle t'_1,\ldots,t'_n\rangle \\ (l,\rho,\kappa,\delta) \leadsto_s (l',\rho',\kappa',\delta') \qquad \upsilon' = \upsilon[\langle l,s\rangle \mapsto \langle t_1,\ldots,t_m\rangle][\langle l',s\rangle \mapsto \langle t'_1,\ldots,t'_n,\langle\rho',\kappa'\rangle\rangle] \end{array}}{\langle \upsilon,\delta\rangle \rightarrow_\tau \langle \upsilon',\delta'\rangle}$$

$$\frac{\begin{array}{c} \upsilon\langle l,s\rangle = \langle t_1,\ldots,t_i,\langle\rho,\kappa\rangle,t_{i+1},\ldots,t_m\rangle \qquad \upsilon\langle l',s\rangle = \langle t'_1,\ldots,t'_n\rangle \\ \upsilon\langle l_s,l_s\rangle = \langle t''_1,\ldots,t''_k\rangle \qquad [\![\,[\,\mathbf{spawn}\,[\,e\,]^{l_s}\,]^{l}_{l'}]\!] \in P \\ \upsilon' = \upsilon[\langle l,s\rangle \mapsto \langle t_1,\ldots,t_m\rangle][\langle l',s\rangle \mapsto \langle t'_1,\ldots,t'_n,\langle\rho[l' \mapsto \text{TRUE}],\kappa\rangle\rangle][\langle l_s,l_s\rangle \mapsto \langle t''_1,\ldots,t''_k,\langle\rho,\langle\rangle\rangle\rangle] \end{array}}{\langle \upsilon,\delta\rangle \rightarrow_\tau \langle \upsilon',\delta\rangle}$$

Figure 6: The definition of the exact transition relation $\rightarrow_\tau$.

Each expression $e \in P$ induces elements of the relation $\leadsto_s$ for each $s \in \mathit{Slabel}$ as follows, where $\mathit{new}(l,s,\theta)$ returns the least i such that $\langle l,s,i\rangle \notin \mathrm{Dom}(\theta)$.

$[\![\,[\,c\,]^{l}_{l'}]\!]$:

$$(l,\rho,\kappa,\delta) \leadsto_s (l',\rho[l' \mapsto c],\kappa,\delta)$$

$[\![\,[\,x\,]^{l}_{l'}]\!]$:

$$(l,\rho,\kappa,\delta) \leadsto_s (l',\rho[l' \mapsto \rho(x)],\kappa,\delta)$$

$[\![\,[\,\lambda x_1\,x_2\,\ldots\,x_n.\,[\,e\,]^{l_f}_{l'_f}\,]^{l}_{l'}]\!]$:

$$(l,\rho,\kappa,\langle\psi,\omega,\theta\rangle) \leadsto_s (l',\rho[l' \mapsto \sigma],\kappa,\langle\psi,\omega,\theta[\sigma \mapsto \rho]\rangle); \quad \sigma = \langle l_f,s,\mathit{new}(l_f,s,\theta)\rangle$$
$$(l'_f,\rho,\langle k_1,\ldots,k_n,\langle l'',\rho'\rangle\rangle,\delta) \leadsto_s (l'',\rho'[l'' \mapsto \rho(l'_f)],\langle k_1,\ldots,k_n\rangle,\delta)$$

$[\![\,[\,p([\,e_1\,]^{l_1}_{l'_1},\ldots,[\,e_n\,]^{l_n}_{l'_n})\,]^{l}_{l'}]\!]$:

$$(l,\rho,\kappa,\delta) \leadsto_s (l_1,\rho,\kappa,\delta)$$
$$(l'_i,\rho,\kappa,\delta) \leadsto_s (l_{i+1},\rho,\kappa,\delta); \quad 1 \le i \le n-1$$
$$(l'_n,\rho,\kappa,\delta) \leadsto_s (l',\rho[l' \mapsto d],\kappa,\delta'); \quad \text{if } \langle l,s,\langle\rho(l'_1),\ldots,\rho(l'_n)\rangle,\delta\rangle \hookrightarrow_p \langle d,\delta'\rangle$$

$[\![\,[\,[\,e_1\,]^{l_1}_{l'_1}\,([\,e_2\,]^{l_2}_{l'_2}\,[\,e_3\,]^{l_3}_{l'_3}\,\ldots\,[\,e_n\,]^{l_n}_{l'_n})\,]^{l}_{l'}]\!]$:

$$(l,\rho,\kappa,\delta) \leadsto_s (l_1,\rho,\kappa,\delta)$$
$$(l'_i,\rho,\kappa,\delta) \leadsto_s (l_{i+1},\rho,\kappa,\delta); \quad 1 \le i \le n-1$$
$$(l'_n,\rho,\langle k_1,\ldots,k_n\rangle,\langle\psi,\omega,\theta\rangle) \leadsto_s (l_f,\theta\,(\rho(l'_1))[x_{i-1} \mapsto \rho(l'_i)],\langle k_1,\ldots,k_n,\langle\, l',\rho\rangle\rangle,\delta);$$
$$2 \le i \le n$$
$$l_f = (\rho(l'_1))\downarrow_1, \quad [\![\lambda x_1\,x_2\,\ldots\,x_{n-1}.\,[\,e\,]^{l_f}]\!] \in P$$

$[\![\,[\,\mathbf{if}\,[\,e_1\,]^{l_1}_{l'_1}\;\mathbf{then}\,[\,e_2\,]^{l_2}_{l'_2}\;\mathbf{else}\,[\,e_3\,]^{l_3}_{l'_3}\,]^{l}_{l'}]\!]$:

$$(l,\rho,\kappa,\delta) \leadsto_s (l_1,\rho,\kappa,\delta)$$
$$(l'_1,\rho,\kappa,\delta) \leadsto_s (l_2,\rho,\kappa,\delta); \quad \rho(l'_1) = \text{TRUE}$$
$$(l'_1,\rho,\kappa,\delta) \leadsto_s (l_3,\rho,\kappa,\delta); \quad \rho(l'_1) = \text{FALSE}$$
$$(l'_i,\rho,\kappa,\delta) \leadsto_s (l',\rho[l' \mapsto \rho(l'_i)],\kappa,\delta); \quad i \in \{2,3\}$$

$[\![\,[\,\mathbf{letrec}\,\ldots\,y_i = \lambda x_1\,x_2\,\ldots\,x_n.\,[\,e_i\,]^{l_i}_{l'_i}\,\ldots\,\mathbf{in}\,[\,e\,]^{l_e}_{l'_e}\,]^{l}_{l'}]\!]$:

$$(l,\rho,\kappa,\langle\psi,\omega,\theta\rangle) \leadsto_s (l_e,\rho',\kappa,\langle\psi,\omega,\theta[\sigma_i \mapsto \rho']\rangle); \quad \sigma_i = \langle l_i,s,\mathit{new}(l_i,s,\theta)\rangle, \; \rho' = \rho[y_i \mapsto \sigma_i]$$
$$(l'_e,\rho,\kappa,\delta) \leadsto_s (l',\rho[\,l' \mapsto \rho(\,l'_e)],\kappa,\delta)$$
$$(l'_i,\rho,\langle k_1,\ldots,k_n,\langle l'',\rho'\rangle\rangle,\delta) \leadsto_s (l'',\rho'[l'' \mapsto \rho(l'_i)],\langle k_1,\ldots,k_n\rangle,\delta)$$

Figure 7: The definition of $\leadsto_s$.

Let $new(l, s, \psi)$ and $new(l, s, \omega)$ be defined similarly to $new(l, s, \theta)$ in Fig 7.

$$\begin{array}{rcll}
\langle l, s, \langle c_1, c_2 \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{+} & \langle c_1 + c_2, \langle \psi, \omega, \theta \rangle \rangle & \\
\langle l, s, \langle d_1, d_2 \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{\text{cons}} & \langle \sigma, \langle \psi, \omega[\sigma \mapsto \langle d_1, d_2 \rangle], \theta \rangle \rangle; & \sigma = \langle l, s, new(l, s, \omega) \rangle \\
\langle l, s, \langle \sigma \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{\text{car}} & \langle (\omega(\sigma)) \downarrow_1, \langle \psi, \omega, \theta \rangle \rangle & \\
\langle l, s, \langle \sigma \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{\text{cdr}} & \langle (\omega(\sigma)) \downarrow_2, \langle \psi, \omega, \theta \rangle \rangle & \\
\langle l, s, \langle \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{\text{mkloc}} & \langle \sigma, \langle \psi[\sigma \mapsto \text{UNBOUND}], \omega, \theta \rangle \rangle; & \sigma = \langle l, s, new(l, s, \psi) \rangle \\
\langle l, s, \langle \sigma, d \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{\text{write}} & \langle d, \langle \psi[\sigma \mapsto d], \omega, \theta \rangle \rangle & \\
\langle l, s, \langle \sigma \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{\text{read}} & \langle \psi(\sigma), \langle \psi, \omega, \theta \rangle \rangle; & \text{if } (\delta \downarrow_1)(\sigma) \neq \text{UNBOUND} \\
\langle l, s, \langle \sigma \rangle, \langle \psi, \omega, \theta \rangle \rangle & \hookrightarrow_{\text{remove}} & \langle \psi(\sigma), \langle \psi[\sigma \mapsto \text{UNBOUND}], \omega, \theta \rangle \rangle; & \text{if } \psi(\sigma) \neq \text{UNBOUND}
\end{array}$$

Figure 8: The exact semantics of some primitive operations.

$$\begin{array}{rcrcl}
\widehat{\phi} & \in & \widehat{State} & = & \widehat{ThreadMap} \times \widehat{Heap} \\
\widehat{\upsilon} & \in & \widehat{ThreadMap} & = & (Label \times Slabel \rightarrow \widehat{Env}) \\
\widehat{\rho}, \tilde{\rho} & \in & \widehat{Env} & = & (Var + Label \rightarrow \widehat{D}) \\
 & & & & +\{\text{ABSENT}\} \\
\widehat{\delta} & \in & \widehat{Heap} & = & \widehat{Locs} \times \widehat{Pairs} \\
\widehat{\psi} & \in & \widehat{Locs} & = & \widehat{Ptr} \rightarrow \widehat{D} \\
\widehat{\omega} & \in & \widehat{Pairs} & = & \widehat{Ptr} \rightarrow \widehat{D} \times \widehat{D} \\
\widehat{d} & \in & \widehat{D} & = & \mathbf{Pow}(\{\text{CONST}\} + \widehat{Ptr}) \\
\widehat{\sigma} & \in & \widehat{Ptr} & = & Label \times Slabel
\end{array}$$

Figure 9: An abstraction of $\mathbf{Pow}(State)$.

We express the relationship between the exact and approximate domains by defining an abstraction function $\alpha \in \mathbf{Pow}(State) \rightarrow \widehat{State}$. Its definition is given in Fig. 10. If $\alpha(\Phi) = \langle \widehat{\upsilon}, \widehat{\delta} \rangle$ then $\widehat{\upsilon}\langle l, s \rangle = \text{ABSENT}$ iff there is no thread in a configuration in Φ at program label l and spawn point s. Intuitively, the transition function in Section 6.2 will "raise" environments above ABSENT as new $\langle$*program-label*, *spawn-point*$\rangle$ pairs are reached. In other words, environments at a given $\langle$*program-label*, *spawn-point*$\rangle$ that are never accessed by any thread are mapped to ABSENT by the thread map approximation.

The concretization function on abstract states is defined thus:

$$\gamma(\widehat{\phi}) = \bigcup \{\Phi \in \mathbf{Pow}(State) \mid \alpha(\Phi) \sqsubseteq \widehat{\phi}\}$$
$$\{\phi \in State \mid \alpha(\{\phi\}) \sqsubseteq \widehat{\phi}\}$$

6.2 Transition Function

The transition function $\widehat{F}$ specific to a program $P \in Exp$ is defined as

$$\widehat{F}(\widehat{\phi}) = (\lambda \langle l, s \rangle . \widehat{F}_{l,s}(\widehat{\phi}), \widehat{F}_{\delta}(\widehat{\phi})).$$

The definition of $\widehat{F}_{l,s} \in \widehat{State} \rightarrow \widehat{Env}$ is given in Fig. 12 and the definition of $\widehat{F}_{\delta} \in \widehat{State} \rightarrow \widehat{Heap}$ is given in Fig. 13.

Fig. 12, the definition of $\widehat{F}_{l,s}$, is the main part of the transition function $\widehat{F}$. The $\bowtie$ function for environments is similar to a join, but is used to prevent unnecessary accumulation of environment information in $\langle$*program-label*, *spawn-point*$\rangle$ pairs that are never reached.

$$\begin{array}{rcl}
\widehat{\mathcal{P}}\langle +, l, s, \langle \widehat{d}_1, \widehat{d}_2 \rangle, \widehat{\delta} \rangle & = & \{\text{CONST}\} \\
\widehat{\mathcal{P}}\langle \mathbf{cons}, l, s, \langle \widehat{d}_1, \widehat{d}_2 \rangle, \widehat{\delta} \rangle & = & \{\langle l, s \rangle\} \\
\widehat{\mathcal{P}}\langle \mathbf{car}, l, s, \langle \widehat{d} \rangle, \langle \widehat{\psi}, \widehat{\omega} \rangle \rangle & = & \bigsqcup_{\widehat{\sigma} \in \widehat{d}} \widehat{\omega}(\widehat{\sigma}) \downarrow_1 \\
\widehat{\mathcal{P}}\langle \mathbf{cdr}, l, s, \langle \widehat{d} \rangle, \langle \widehat{\psi}, \widehat{\omega} \rangle \rangle & = & \bigsqcup_{\widehat{\sigma} \in \widehat{d}} \widehat{\omega}(\widehat{\sigma}) \downarrow_2 \\
\widehat{\mathcal{P}}\langle \mathbf{mk\text{-}loc}, l, s, \langle \rangle, \widehat{\delta} \rangle & = & \{\langle l, s \rangle\} \\
\widehat{\mathcal{P}}\langle \mathbf{write}, l, s, \langle \widehat{d}_1, \widehat{d}_2 \rangle, \widehat{\delta} \rangle & = & \widehat{d}_2 \\
\widehat{\mathcal{P}}\langle \mathbf{read}, l, s, \langle \widehat{d} \rangle, \langle \widehat{\psi}, \widehat{\omega} \rangle \rangle & = & \bigsqcup_{\widehat{\sigma} \in \widehat{d}} \widehat{\psi}(\widehat{\sigma}) \\
\widehat{\mathcal{P}}\langle \mathbf{remove}, l, s, \langle \widehat{d} \rangle, \langle \widehat{\psi}, \widehat{\omega} \rangle \rangle & = & \bigsqcup_{\widehat{\sigma} \in \widehat{d}} \widehat{\psi}(\widehat{\sigma})
\end{array}$$

Figure 11: The definition of some approximate primitive operations.

The rules for constants, identifiers, and conditionals are straightforward. The value of an abstraction is a pair $\langle l, s \rangle$, where l is the entry label of the abstraction's body, and s is the spawn point in which the abstraction was evaluated. Primitive operations are defined via the auxiliary function $\widehat{\mathcal{P}}$; excerpts from its definition are given in Fig. 11. Note that we do not consider the blocking behavior of **read** and **remove** in the definition of the abstract semantics. Thus, optimizers based on information generated by the interpretation examine $\langle$*program label*, *spawn point*$\rangle$ pairs found in an abstract environment to determine the readers and removers of a given location, and their inter-dependencies.

The rule for the entry of an abstraction deserves elaboration. If an application at spawn point s applies a function $\langle l_f, s' \rangle$, then the body of the function at label l_f *at spawn point* s should include the environment in which $\langle l_f, s' \rangle$ was created extended with bindings for its arguments. This is because only spawns dynamically change spawn points, not function calls. Thus, in the definition of $\tilde{\rho}_{l_f,s}$, the environment is join over the environments of all functions at $\langle l_f, s' \rangle$ that

$$
\begin{array}{rcl}
\alpha(\Phi) & = & \langle \alpha_{\mathrm{ThreadMap}}(\{\upsilon \mid \langle \upsilon, \delta\rangle \in \Phi\}), \alpha_{\mathrm{Heap}}(\{\delta \mid \langle \upsilon, \delta\rangle \in \Phi\})\rangle \\
\alpha_{\mathrm{ThreadMap}}(\Upsilon) & = & \lambda\langle l, s\rangle.\, \alpha_{\mathrm{Env}}(\{\rho \mid \langle \ldots, \langle \rho, \kappa\rangle, \ldots\rangle = \upsilon\langle l, s\rangle,\ \upsilon \in \Upsilon\}) \\
\alpha_{\mathrm{Env}}(E) & = & \begin{cases} \text{ABSENT} & \text{if } E = \emptyset \\ \lambda z.\, \alpha_{\mathrm{D}}(\{\rho(z) \mid \rho \in E\}) & \text{otherwise} \end{cases} \\
\alpha_{\mathrm{Heap}}(\Delta) & = & \langle \alpha_{\mathrm{Locs}}(\{\psi \mid \langle \psi, \omega, \theta\rangle \in \Delta\}), \alpha_{\mathrm{Pairs}}(\{\omega \mid \langle \psi, \omega, \theta\rangle \in \Delta\})\rangle \\
\alpha_{\mathrm{Locs}}(\Psi) & = & \lambda\langle l, s\rangle.\, \alpha_{\mathrm{D}}(\{d \mid \psi \in \Psi,\ d = \psi\langle l, s, i\rangle,\ i \in \mathit{Nat}\}) \\
\alpha_{\mathrm{Pairs}}(\Omega) & = & \lambda\langle l, s\rangle.\, \langle \alpha_{\mathrm{D}}(\{\omega\langle l, s, i\rangle\downarrow_1 \mid \omega \in \Omega,\ i \in \mathit{Nat}\}), \\
& & \quad \alpha_{\mathrm{D}}(\{\omega\langle l, s, i\rangle\downarrow_2 \mid \omega \in \Omega,\ i \in \mathit{Nat}\})\rangle \\
\alpha_{\mathrm{D}}(D) & = & \{\{\text{CONST}\} \mid c \in D\} \cup \{\langle l, s\rangle \mid \langle l, s, i\rangle \in D\}
\end{array}
$$

Figure 10: The abstraction function.

$\widehat{F}_{l,s}\langle\widehat{\upsilon}, \langle\widehat{\psi}, \widehat{\omega}\rangle\rangle = \tilde{\rho}_{l,s}$, where each expression $e \in P$ induces equations for all spawn-point labels $s \in \mathit{Slabel}$ as follows. Let $\widehat{\rho}_{l,s} = \widehat{\upsilon}\langle l, s\rangle$. If there is no equation for some $\tilde{\rho}_{l,s}$, then $\tilde{\rho}_{l,s} = \text{ABSENT}$. Let $\text{ABSENT}(x) = \emptyset$. Define $\widehat{\rho}_1 \bowtie \widehat{\rho}_2 \equiv$ if $\widehat{\rho}_1 = \text{ABSENT}$ then ABSENT else $\widehat{\rho}_1 \sqcup \widehat{\rho}_2$.

$[\![[c]^{l}_{l'}]\!]$:

$\tilde{\rho}_{l',s} = \widehat{\rho}_{l,s} \bowtie [l' \mapsto \{\text{CONST}\}]$

$[\![[x]^{l}_{l'}]\!]$:

$\tilde{\rho}_{l',s} = \widehat{\rho}_{l,s} \bowtie [l' \mapsto \widehat{\rho}_{l,s}(x)]$

$[\![[\lambda x_1\, x_2\, \ldots\, x_n.\, [e]^{l_f}]^{l}_{l'}]\!]$:

$\tilde{\rho}_{l',s} = \widehat{\rho}_{l,s} \bowtie [l' \mapsto \{\langle l_f, s\rangle\}]$

$$\tilde{\rho}_{l_f,s} = \bigsqcup\{\widehat{\rho}_{l,s'} \sqcup [x_i \mapsto \widehat{\rho}_{l_{i+1},s}(l_{i+1})] \mid [\![[e_1]_{l_1} ([e_2]_{l_2}\, [e_3]_{l_3}\, \ldots\, [e_{n+1}]_{l_{n+1}})]\!] \in P,\ \langle l_f, s'\rangle \in \widehat{\rho}_{l_1,s}(l_1)\}\quad 1 \leq i \leq n$$

$[\![p([e_1]^{l_1}_{l'_1}, \ldots, [e_n]^{l_n}_{l'_n})]^{l}_{l'}]\!]$:

$\tilde{\rho}_{l_1,s} = \widehat{\rho}_{l,s}$

$\tilde{\rho}_{l_i,s} = \widehat{\rho}_{l'_{i-1},s};\ 2 \leq i \leq n$

$\tilde{\rho}_{l',s} = \widehat{\rho}_{l'_n,s} \bowtie [l' \mapsto \widehat{\mathcal{P}}\langle p, l, s, \langle\widehat{\rho}_{l'_n,s}(l'_1), \ldots, \widehat{\rho}_{l'_n,s}(l'_n)\rangle, \langle\widehat{\psi}, \widehat{\omega}\rangle\rangle]$

$[\![[[e_1]^{l_1}_{l'_1} ([e_2]^{l_2}_{l'_2}\, [e_3]^{l_3}_{l'_3}\, \ldots\, [e_n]^{l_n}_{l'_n})]^{l}_{l'}]\!]$:

$\tilde{\rho}_{l_1,s} = \widehat{\rho}_{l,s}$

$\tilde{\rho}_{l_i,s} = \widehat{\rho}_{l'_{i-1},s};\ 2 \leq i \leq n$

$\tilde{\rho}_{l',s} = \widehat{\rho}_{l'_n,s} \bowtie \bigsqcup\{[l' \mapsto \widehat{\rho}_{l'_f,s}(l'_f)] \mid [\![\lambda x_1\, x_2\, \ldots\, x_n.\, [e]^{l_f}_{l'_f}]\!] \in P,\ \langle l_f, s'\rangle \in \widehat{\rho}_{l'_n,s}(l'_1)\}$

$[\![[\textbf{if}\ [e_1]^{l_1}_{l'_1}\ \textbf{then}\ [e_2]^{l_2}_{l'_2}\ \textbf{else}\ [e_3]^{l_3}_{l'_3}]^{l}_{l'}]\!]$:

$\tilde{\rho}_{l_1,s} = \widehat{\rho}_{l,s}$

$\tilde{\rho}_{l_i,s} = \widehat{\rho}_{l'_1,s};\ i \in \{2, 3\}$

$\tilde{\rho}_{l',s} = \bigsqcup_{i \in \{2,3\}} (\widehat{\rho}_{l'_i,s} \bowtie [l' \mapsto \widehat{\rho}_{l'_i,s}(l'_i)])$

$[\![[\textbf{letrec}\ \ldots\ y_i = \lambda x_1\, x_2\, \ldots\, x_n.\, [e_i]^{l_i}\ \ldots\ \textbf{in}\ [e]^{l_e}_{l'_e}]^{l}_{l'} :]\!]$

$\tilde{\rho}_{l_e,s} = \widehat{\rho}_{l,s} \bowtie [y_i \mapsto \{\langle l_i, s\rangle\}]$

$\tilde{\rho}_{l',s} = \widehat{\rho}_{l'_e,s} \bowtie [l' \mapsto \widehat{\rho}_{l'_e,s}(l'_e)]$

$\tilde{\rho}_{l_i,s} = \bigsqcup\{\widehat{\rho}_{l_e,s'} \sqcup [x_i \mapsto \widehat{\rho}_{l_{i+1},s}(l_{i+1})] \mid [\![[e_1]_{l_1} ([e_2]_{l_2}\, [e_3]_{l_3}\, \ldots\, [e_{n+1}]_{l_{n+1}})]\!] \in P,\ \langle l, s'\rangle \in \widehat{\rho}_{l_1,s}(l_1)\}$

$[\![[\textbf{spawn}\, [e]^{l_s}]^{l}_{l'}]\!]$:

$\tilde{\rho}_{l',s} = \widehat{\rho}_{l,s} \bowtie [l' \mapsto \{\text{CONST}\}]$

$\tilde{\rho}_{l_s,l_s} = \bigsqcup_{s' \in \mathit{Slabel}} \widehat{\rho}_{l,s'}$

Figure 12: The definition of $\widehat{F}_{l,s}$.

$\widehat{F}_\delta\langle\widehat{v},\langle\widehat{\psi},\widehat{\omega}\rangle\rangle = \langle\widehat{\psi}',\widehat{\omega}'\rangle$, where $\widehat{\rho}_{l,s} = \widehat{v}\langle l,s\rangle$ and

$$\begin{aligned}
\widehat{\psi}' &= \widehat{\psi} \sqcup \bigsqcup\{[\langle l,s\rangle \mapsto \{\textsc{const}\}] \mid [\![[\,(\textbf{mk-loc})\,]^l]\!] \in P,\ \widehat{\rho}_{l,s} \neq \textsc{absent}\} \\
&\quad \sqcup \bigsqcup\{[\widehat{\sigma} \mapsto \widehat{\rho}_{l_2,s}(l_2)] \mid [\![\textbf{write}([\,e_1\,]_{l_1},[\,e_2\,]_{l_2})]\!] \in P,\ \widehat{\sigma} \in \widehat{\rho}_{l_1,s}(l_1),\ s \in \mathit{Slabel}\} \\
\widehat{\omega}' &= \widehat{\omega} \sqcup \bigsqcup\{[\langle l,s\rangle \mapsto (\widehat{\rho}_{l_1,s}(l_1),\widehat{\rho}_{l_2,s}(l_2))] \mid [\![[\,\textbf{cons}([\,e_1\,]_{l_1},[\,e_2\,]_{l_2})\,]^l]\!] \in P,\ s \in \mathit{Slabel}\}
\end{aligned}$$

Figure 13: The definition of $\widehat{F}_\delta$.

$$\begin{array}{l}
\textbf{let}\ f\ =\ \lambda\,x. \\
\qquad\qquad [\textbf{let}\ g\ =\ \lambda\,y.[\,\textbf{write}(y,0)\,]^{l_g} \\
\qquad\qquad\ \ \textbf{in}\ \textbf{write}(x,g)]^{l_f} \\
\textbf{in begin} \\
\quad \textbf{spawn}[\ \textbf{let}\ m = [\,(\textbf{mk-loc})\,]^{l_m} \\
\qquad\qquad\quad b = [\,(\textbf{mk-loc})\,]^{l_b} \\
\qquad\quad \textbf{in begin} \\
\qquad\qquad [f]^{l_1}\ (m) \\
\qquad\qquad \textbf{read}(m)\ (b) \\
\qquad\quad \textbf{end} \\
\quad\ ]^{s_1} \\
\quad \textbf{spawn}[\ \textbf{let}\ n = [\,(\textbf{mk-loc})\,]^{l_n} \\
\qquad\qquad\quad c = [\,(\textbf{mk-loc})\,]^{l_c} \\
\qquad\quad \textbf{in begin} \\
\qquad\qquad [f]^{l_2}\ (n) \\
\qquad\qquad \textbf{remove}(n)\ (c) \\
\qquad\quad \textbf{end} \\
\quad\ ]^{s_2} \\
\textbf{end}
\end{array}$$

Figure 14: Constructing approximate environments is complicated by the presence of higher-order procedures and dynamically instantiated processes.

Program Labels	Spawn Labels	
	s_1	s_2
l_1	$f \mapsto \{\langle l_f, s_0\rangle\}$	ABSENT
l_2	ABSENT	$f \mapsto \{\langle l_f, s_0\rangle\}$
l_f	$x \mapsto \{\langle l_m, s_1\rangle\}$	$x \mapsto \{\langle l_n, s_2\rangle\}$
l_g	$x \mapsto \{\langle l_m, s_1\rangle\}$ $y \mapsto \{\langle l_b, s_1\rangle\rangle\}$	$x \mapsto \{\langle l_n, s_2\rangle\rangle\}$ $y \mapsto \{\langle l_c, s_2\rangle\rangle\}$

Heap
$\langle l_m, s_1\rangle \mapsto \{\langle l_g, s_1\rangle\}$
$\langle l_n, s_2\rangle \mapsto \{\langle l_g, s_2\rangle\}$
$\langle l_b, s_1\rangle \mapsto \{\textsc{const}\}$
$\langle l_c, s_2\rangle \mapsto \{\textsc{const}\}$

Figure 15: Salient binding and store information derived for the program shown in Fig.14.

are applied at spawn point s.

The rule for the exit of an application is also non-trivial. Consider the function $\lambda x.[\,e\,]_l$. The result of this function is placed in the environment via program label l, but there is a different environment, and thus potentially a different result, for every spawn label. The environment just after an application at spawn point s is therefore the environment just before the application, extended with the join of the result *at spawn point s* of each function that may have been applied.

The subtlety of the abstraction and application rules is due to the interaction of higher-order procedures and dynamic process instantiation. To illustrate this point, consider the program fragment shown in Fig. 14. In this example, g is closed over different environments in the two threads generated within the **let**-body. It is useful to avoid collapsing these environments. In addition, it is also useful not to collapse the approximate values to which y is bound across thread boundaries. Relevant binding information derived for this program is shown in Fig. 15. This information reveals that the two applications of g made by threads do not write to the same location; thus, **read**'s executed by threads created at spawn points instantiated at s_1 are guaranteed not to block because of the **remove** operation executed by threads instantiated at s_2.

As described in Section 5.1, regardless of the spawn point during execution of **spawn** $[\,e\,]^s$, the spawn point during execution of e is s. Thus, in the approximate semantics, environments for all possible spawn points at this entry label s must be joined together.

7 Correctness of the Interpretation

Lemma 1 *$\widehat{F}$ is monotonic.* □

Lemma 2 *$\widehat{F}$ is an upper approximation of F (i.e., $\alpha \circ F \sqsubseteq \widehat{F} \circ \alpha$).* □

Recall that we wish to compute an approximation of the strongest global invariant of a program $[\,P\,]^l \in \mathit{Exp}$. As described in Section 5, the strongest global invariant is expressible as a fixpoint $\mathit{lfp}(F,\Phi_0)$, where Φ_0 is the set of initial configurations of P. Define $\widehat{\phi}_0 = \langle[\langle l,s_0\rangle \mapsto \lambda z.\,\emptyset], \bot_{\widehat{\mathrm{Heap}}}\rangle$.

Lemma 3 *$\mathit{lfp}(\widehat{F},\widehat{\phi}_0)$ is an upper approximation to $\mathit{lfp}(F,\Phi_0)$ (i.e., $\mathit{lfp}(F,\Phi_0) \subseteq \gamma(\mathit{lfp}(\widehat{F},\widehat{\phi}_0))$).*

Proof: An initial configuration of P will have an empty heap and a single thread at program label l and implicit spawn point s_0 with an empty environment. Therefore, $\alpha(\Phi_0) = \widehat{\phi}_0$. By Lemma 2, $\widehat{F}$ is a correct upper approximation to F. Thus, $\lambda\widehat{\phi}.\,\widehat{\phi}_0 \sqcup \widehat{F}(\widehat{\phi})$ is a correct upper approximation to $\lambda\Phi.\,\Phi_0 \cup F(\Phi)$. By the definition of *lfp* and [9], $\mathit{lfp}(\widehat{F}, \widehat{\phi}_0)$ is thus a correct upper approximation to $\mathit{lfp}(F, \Phi_0)$. □

Lemma 4 $\mathit{lfp}(\widehat{F}, \widehat{\phi}_0)$ *is computable in time polynomial in the size of* P.

Proof: $\mathit{lfp}(\widehat{F}, \widehat{\phi}_O)$ can be computed by a standard iterative fixpoint computation that uses $\widehat{F}$ to monotonically move up the $\widehat{\mathit{State}}$ lattice. We show that the height of $\widehat{\mathit{State}}$ is O(*poly*(n)), and that a single iteration in the fixpoint, *i.e.,* the computation of $\widehat{F}$, requires $O(\mathit{poly}(n))$ time. Because $\widehat{F}$ is monotonic, $\mathit{lfp}(\widehat{F}, \widehat{\phi}_O)$ can be computed in $O(\mathit{poly}(n))$ time.

Let n be the size of P. We can consider $\widehat{F}$ to be defined over a finite subset $\widehat{\mathit{State}_P}$ of $\widehat{\mathit{State}}$, defined as in Fig. 9, but where *Var*, *Label*, and *Slabel* are replaced by finite subsets Var_P (the set of variables appearing in P), Label_P (the set of labels appearing in P), and Slabel_P (the set of spawn labels appearing in P), respectively. Var_P, Label_P, and Slabel_P are all of size $O(n)$.

Because Label_P and Slabel_P are size $O(n)$, the size of $\widehat{\mathit{Ptr}}$ is $O(\mathit{poly}(n))$. Hence the height of $\widehat{D}$ is $O(\mathit{poly}(n))$, and thus the height of $\widehat{\mathit{Locs}}$ and $\widehat{\mathit{Pairs}}$ is $O(\mathit{poly}(n))$. It follows that the height of $\widehat{\mathit{Heap}}$ is therefore also $O(\mathit{poly}(n))$. Because the size of Var_P and Label_P is $O(n)$ and the height of $\widehat{D}$ is $O(\mathit{poly}(n))$, the height of $\widehat{\mathit{Env}}$ is $O(\mathit{poly}(n))$. Moreover, since the size of $\mathit{Label}_P \times \mathit{Slabel}_P$ is $O(\mathit{poly}(n))$, we know that the height of $\widehat{\mathit{ThreadMap}}$ is $O(\mathit{poly}(n))$. Thus, the height of $\widehat{\mathit{State}}$ is $O(\mathit{poly}(n))$.

A single computation of $\widehat{F}$ computes $O(\mathit{poly}(n))$ environments and heap entries since there are $O(n)$ equations for each spawn point in Slabel_P. The computation of a new entry requires computing the join of a polynomial number of elements of $\widehat{D}$. But, because the height of $\widehat{D}$ is $O(\mathit{poly}(n))$, computing the join of two elements in $\widehat{D}$ (essentially a set union) takes time $O(\mathit{poly}(n))$. Hence the time complexity of computing a new environment or heap entry is $O(\mathit{poly}(n))$.

□

8 Conclusions

Applying abstract interpretation techniques to expressive parallel symbolic languages appears to be a promising avenue for future investigation. Although the interpretation given here offers opportunities for data locality and thread mapping optimizations in multi-threaded higher-order languages, there are important extensions to the framework that may enable other kinds of useful optimizations. For example, constructing refined representations of abstract locations and abstract threads capable of distinguishing their different instances within a given context would more easily enable optimizations such as test-for-presence, lock elimination, safe inlining, etc.. In addition, constructing an abstraction of the exact state sensitive to the blocking semantics of **read** and **remove** might facilitate other optimizations concerned with compile-time scheduling and mapping. We intend to pursue such extensions as a focus of future research.

The implementation of the analysis is currently written in T [26], a dialect of Scheme. We have also implemented a parallel version of the abstract interpreter [33] described here that runs on top of Sting [20]. Abstract interpretation is a good domain for parallel symbolic computing because the internal structure of abstract interpreters typically contains many concurrently evaluatable components. Parallel implementations of abstract interpreters may thus permit experimentation with other kinds of useful, but computationally expensive, analyses.

Acknowledgments

Thanks to Christopher Colby and Henry Cejtin for many useful discussions and comments.

References

[1] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Paul Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the ACM Symposium on Functional Programming and Computer Architecture*, pages 538–568, August 1991. Published as Springer Verlag LNCS 523.

[3] Adrienne Bloss, Paul Hudak, and Jonathan Young. An Optimizing Compiler for a Modern Functional Language. *The Computer Journal*, 2(32), April 1989.

[4] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[5] Jyh-Herng Chow, June 1993. Personal Communication.

[6] Jyh-Herng Chow and Williams Ludwell Harrison III. Compile Time Analysis of Parallel Programs that Share Memory. In *19^{th} ACM Symposium on Principles of Programming Languages*, January 1992.

[7] William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[8] Eric C. Cooper and J. Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[9] Patrick. Cousot. Semantic Foundations of Program Analysis. In *Program Flow Analysis: Theory and Foundation*, pages 303–342. Prentice-Hall, 1981.

[10] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. In *ACM 4^{th} Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[11] Saumya Debray and David Warren. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–451, 1989.

[12] Alain Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. In *17^{th} ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[13] R. Gabriel and J. McCarthy. Queue-Based Multi-Processing Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 25–44, August 1984.

[14] Allan Gottlieb, B. Lubachevsky, and Larry Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[15] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[16] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, June 1989.

[17] Paul Hudak and Jonathan Young. A Collecting Interpretation of Expressions. *ACM Transactions on Programming Languages and Systems*, pages 269–290, April 1991.

[18] Williams Ludwell Harrison III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation: an International Journal*, 2(3/4):179–396, 1989.

[19] T. Ito and R.H Halstead, editors. *Parallel Lisp: Languages and Systems*. Springer-Verlag, 1989. LNCS 441.

[20] Suresh Jagannathan and James Philbin. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the 1992 Conf. on Lisp and Functional Programming*, pages 345–357, June 1992.

[21] Monica Lam and Martin Rinard. Coarse-Grained Parallel Programming in Jade. In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 94–105, 1991.

[22] John Lucassen and David Gifford. Polymorphic Effect Systems. In *15^{th} ACM Symposium on Principles of Programming Languages*, January 1988.

[23] John Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91*, pages 24–33, 1991.

[24] N. Mercouroff. An Algorithm for Analyzing Communicating Processes. In *Mathematical Foundations of Programming Semantics*. Springer-Verlag, 1991.

[25] J. Gregory Morrisett and Andrew Tolmach. Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey. In *Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 198–207, 1993.

[26] Jonathan A. Rees and Norman I. Adams. T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

[27] John Reif and Scott Smolka. Data Flow Analysis of Distributed Communicating Processes. *International Journal of Parallel Programming*, 19(1):1–30, 1990.

[28] John Reppy. CML: A Higher-Order Concurrent Language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–306, June 1991.

[29] Ehud Shapiro, editor. *Concurrent Prolog : Collected Papers*. MIT Press, 1987. Volumes 1 and 2.

[30] Olin Shivers. Data-flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation*. MIT Press, 1990.

[31] Olin Shivers. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, 1991.

[32] Guy L. Steele Jr. Making Asynchronous Paralleism Safe for the World. In *ACM 17^{th} Symposium on Principles of Programming Languages*, pages 218–231, January 1990.

[33] Stephen Weeks, Suresh Jagannathan, and James Philbin. A Concurrent Abstract Interpreter. *Lisp and Symbolic Computation*. To Appear.