A family of abstract interpretations for static analysis of concurrent higher-order programs

Matthew Might and David Van Horn

University of Utah and Northeastern University
might@cs.utah.edu and dvanhorn@ccs.neu.edu
http://matt.might.net/ and http://lambda-calcul.us/

Abstract. We develop a framework for computing two foundational analyses for *concurrent* higher-order programs: (control-)flow analysis (CFA) and may-happen-in-parallel analysis (MHP). We pay special attention to the unique challenges posed by the unrestricted mixture of first-class continuations and dynamically spawned threads. To set the stage, we formulate a concrete model of concurrent higher-order programs: the P(CEK*)S machine. We find that the systematic abstract interpretation of this machine is capable of computing both flow and MHP analyses. Yet, a closer examination finds that the precision for MHP is poor. As a remedy, we adapt a shape analytic technique—singleton abstraction—to dynamically spawned threads (as opposed to objects in the heap). We then show that if MHP analysis is not of interest, we can substantially accelerate the computation of flow analysis alone by collapsing thread interleavings with a second layer of abstraction.

1 Higher-order is hard; concurrency makes it harder

The next frontier in static reasoning for higher-order programs is concurrency. When unrestricted concurrency and higher-order computation meet, their challenges to static reasoning reinforce and amplify one another.

Consider the possibilities opened by a mixture of dynamically created threads and first-class continuations. Both pose obstacles to static analysis by themselves, yet the challenge of reasoning about a continuation created in one thread and invoked in another is substantially more difficult than the sum of the individual challenges.

We respond to the challenge by (1) constructing the P(CEK*)S machine, a nondeterministic abstract machine that concretely and fully models higher-orderness and concurrency; and then (2) systematically deriving abstract interpretations of this machine to enable the sound and meaningful flow analysis of concurrent higher-order programs.

Our first abstract interpretation creates a dual hierarchy of flow and may-happen-in-parallel (MHP) analyses parameterized by context-sensitivity and the granularity of an abstract partition among threads. The context-sensitivity knob tunes flow-precision as in Shivers's k-CFA [21]. The partition among threads tunes the precision of MHP analysis, since it controls the mapping of concrete threads onto abstract threads. To improve the precision of MHP analysis, our second abstract interpretation introduces shape analytic concepts—chiefly, singleton cardinality analysis—but it applies them to discover the "shape" of threads rather than the shape of objects in the heap. The final abstract interpretation accelerates the computation of flow analysis (at the cost of MHP analysis) by inflicting a second abstraction that soundly collapses all thread interleavings together.

1.1 Challenges to reasoning about higher-order concurrency

The combination of higher-order computation and concurrency introduces design patterns that challenge conventional static reasoning techniques.

Challenge: Optimizing futures Futures are a popular means of enabling parallelism in functional programming. Expressions marked `future` are computed in parallel with their own continuation. When that value reaches a point of strict evaluation, the thread of the continuation joins with the thread of the future.

Unfortunately, the standard implementation of futures [5] inflicts substantial costs on sequential performance: that implementation transforms `(future` e`)` into `(spawn` e`)`, and all strict expressions into conditionals and thread-joins. That is, if the expression e' is in a strict evaluation position, then it becomes:

```
(let ([$t e′]) (if (thread? $t) (join $t) $t))
```

Incurring this check at all strict points is costly. A flow analysis that works for concurrent programs would find that most expressions can never evaluate to future value, and thus, need not incur such tests.

Challenge: Thread cloning/replication The higher-order primitive `call/cc` captures the current continuation and passes it as a first-class value to its argument. The primitive `call/cc` is extremely powerful—a brief interaction between `spawn` and `call/cc` effortlessly expresses thread replication:

```
(call/cc (lambda (cc) (spawn (cc #t)) #f))
```

This code captures the current continuation, spawns a new thread and replicates the spawning thread in the spawned thread by invoking that continuation. The two threads can be distinguished by the return value of `call/cc`: the replicant returns true and the original returns false.

Challenge: Thread metamorphosis Consider a web server in which continuations are used to suspend and restore computations during interactions with the client [18]. Threads "morph" from one kind of thread (an interaction thread or a worker thread) to another by invoking continuations. The `begin-worker` continuation metamorphizes the calling thread into a worker thread:

```
(define become-worker
 (let ([cc (call/cc (lambda (cc) (cc cc)))])
  (cond
   [(continuation? cc)       cc]
   [else                     (handle-next-request)
                             (become-worker #t)])))
```

The procedure `handle-next-request` checks whether the request is the resumption of an old session, and if so, invokes the continuation of that old session:

```
(define (handle-next-request)
 (define request (next-request))
 (atomic-hash-remove! (session-id request)
  (lambda (session-continuation)
    (define answer (request->answer request))
    (session-continuation answer))
  (lambda () (start-new-session request))))
```

When a client-handling thread needs data from the client, it calls `read-from-client`, it associates the current continuation to the active session, piggy-backs a request to the client on an outstanding reply and the metamorphizes into a worker thread to handle other incoming clients:

```
(define (read-from-client session)
 (call/cc (lambda (cc)
  (atomic-hash-set! sessions (session-id session) cc)
  (reply-to session))
  (become-worker #t)))
```

2 P(CEK*)S: An abstract machine model of concurrent, higher-order computation

In this section, we define a P(CEK*)S machine—a CESK machine with a pointer refinement that allows concurrent threads of execution. It is directly inspired by the sequential abstract machines in Van Horn and Might's recent work [23]. Abstract interpretations of this machine perform both flow and MHP analysis for concurrent, higher-order programs.

The language modeled in this machine (Figure 1) is A-Normal Form lambda calculus [9] augmented with a core set of primitives for multithreaded programming. For concurrency, it features an atomic compare-and-swap operation, a spawn form to create a thread from an expression and a join operation to wait for another thread to complete. For higher-order computation, it features closures and first-class continuations. A closure is a first-class procedure constructed by pairing a lambda term with an environment that fixes the meaning of its free variables. A continuation reifies the sequential control-flow for the remainder of the thread as a value; when a continuation is "invoked," it restores that control-flow. Continuations may be invoked an arbitrary number of times, and at any time since their moment of creation.

$$
\begin{array}{rcl}
e \in \mathsf{Exp} & ::= & \texttt{(let ((}v\ \mathit{cexp}\texttt{))}\ e\texttt{)} \\
& | & \mathit{cexp} \\
& | & æ \\
\mathit{cexp} \in \mathsf{CExp} & ::= & \texttt{(}f\ æ_1 \ldots æ_n\texttt{)} \\
& | & \texttt{(callcc}\ æ\texttt{)} \\
& | & \texttt{(set!}\ v\ æ_{\text{value}}\texttt{)} \\
& | & \texttt{(if}\ æ\ \mathit{cexp}\ \mathit{cexp}\texttt{)} \\
& | & \texttt{(cas}\ v\ æ_{\text{old}}\ æ_{\text{new}}\texttt{)} \\
& | & \texttt{(spawn}\ e\texttt{)} \\
& | & \texttt{(join}\ æ\texttt{)} \\
f, æ \in \mathsf{AExp} & ::= & \mathit{lam} \mid v \mid n \mid \texttt{\#f} \\
\mathit{lam} \in \mathsf{Lam} & ::= & \texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ e\texttt{)}
\end{array}
$$

Fig. 1. ANF lambda-calculus augmented with a core set of primitives for concurrency

2.1 P(CEK*)S: A concrete state-space

A concrete state of execution in the P(CEK*)S machine contains a set of threads plus a shared store. Each thread is a context combined with a thread id. A context contains the current expression, the current environment, an address pointing to the current continuation, and a thread history:

$$
\begin{array}{rl}
\varsigma \in \Sigma = & \mathit{Threads} \times \mathit{Store} \\
T \in \mathit{Threads} = & \mathcal{P}\left(\mathit{Context} \times \mathit{TID}\right) \\
c \in \mathit{Context} = & \mathsf{Exp} \times \mathit{Env} \times \mathit{Addr} \times \mathit{Hist} \\
\rho \in \mathit{Env} = & \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\kappa \in \mathit{Kont} = & \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} \times \mathit{Addr} + \{\mathbf{halt}\} \\
h \in \mathit{Hist} & \text{contains records of thread history} \\
\sigma \in \mathit{Store} = & \mathit{Addr} \to D \\
d \in D = & \mathit{Value} \\
\mathit{val} \in \mathit{Value} = & \mathit{Clo} + \mathit{Bool} + \mathit{Num} + \mathit{Kont} + \mathit{TID} + \mathit{Addr} \\
\mathit{clo} \in \mathit{Clo} = & \mathsf{Lam} \times \mathit{Env} \\
a \in \mathit{Addr} & \text{is an infinite set of addresses} \\
t \in \mathit{TID} & \text{is an infinite set of thread ids.}
\end{array}
$$

The P(CEK*)S machine allocates continuations in the store; thus, to add first-class continuations, we have first-class addresses. Under abstraction, program history determines the context-sensitivity of an individual thread. To allow context-sensitivity to be set as external parameter, we'll leave program history opaque. (For example, to set up a k-CFA-like analysis, the program history would be the sequence of calls made since the start of the program.) To parameterize the precision of MHP analysis, the thread ids are also opaque.

2.2 P(CEK*)S: A factored transition relation

Our goal is to factor the semantics of the P(CEK*)S machine, so that one can drop in a classical CESK machine to model sequential language features. The abstract interpretation maintains the same factoring, so that existing analyses of higher-order programs may be "plugged into" the framework for handling concurrency. The relation $(\Rightarrow)$ models concurrent transition, and the relation $(\rightarrow)$ models sequential transition:

$$\begin{aligned}(\Rightarrow) &\subseteq \Sigma \times \Sigma \\ (\rightarrow) &\subseteq (\mathit{Context} \times \mathit{Store}) \times (\mathit{Context} \times \mathit{Store})\end{aligned}$$

For instance, the concurrent transition relation invokes the sequential transition relation to handle `if`, `set!`, `cas`, `callcc` or procedure call:[1]

$$\frac{(c, \sigma) \rightarrow (c', \sigma')}{(\{(c,t)\} \uplus T, \sigma) \Rightarrow (\{(c',t)\} \cup T, \sigma')}$$

Given a program e, the injection function $\mathcal{I} : \mathsf{Exp} \rightarrow \mathit{State}$ creates the initial machine state:

$$\mathcal{I}(e) = (\{((e, [], a_{\mathbf{halt}}, h_0), t_0)\}, [a_{\mathbf{halt}} \mapsto \mathbf{halt}]),$$

where t_0 is the distinguished initial thread id, h_0 is a blank history and $a_{\mathbf{halt}}$ is the distinguished address of the **halt** continuation. The meaning of a program e is the (possibly infinite) set of states reachable from the initial state:

$$\{\varsigma : \mathcal{I}(e) \Rightarrow^* \varsigma\}.$$

Sequential transition example: `callcc` There are ample resources dating to Felleisen and Friedman [6] detailing the transition relation of a CESK machine. For a recent treatment that covers both concrete and abstract transition, see Van Horn and Might [23]. Most of the transitions are straightforward, but in the interest of more self-containment, we review the `callcc` transition:

$$\begin{aligned}(\overbrace{([\![\texttt{(callcc}\ æ\texttt{)}]\!], \rho, a_\kappa, h)}^{c}, \sigma) &\Rightarrow ((e, \rho'', a_\kappa, h'), \sigma'), \text{ where} \\ h' &= \mathit{record}(c, h) \\ ([\![\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}]\!], \rho') &= \mathcal{E}(æ, \rho, \sigma) \\ a &= \mathit{alloc}(v, h') \\ \rho'' &= \rho'[v \mapsto a] \\ \sigma' &= \sigma[a \mapsto a_\kappa].\end{aligned}$$

[1] The transition for `cas` is "sequential" in the sense that its action is atomic.

The atomic evaluation function $\mathcal{E} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \rightharpoonup D$ maps an atomic expression to a value in the context of an environment and a store; for example:

$$\mathcal{E}(v, \rho, \sigma) = \sigma(\rho(v))$$
$$\mathcal{E}(\mathit{lam}, \rho, \sigma) = (\mathit{lam}, \rho).$$

(The notation $f[x \mapsto y]$ is functional extension: the function identical to f, except that x now yields y instead of $f(x)$.)

2.3 A shift in perspective

Before proceeding, it is worth shifting the formulation so as to ease the process of abstraction. For instance, the state-space is well-equipped to handle a finite abstraction over addresses, since we can promote the range of the store to *sets* of values. This allows multiple values to live at the same address once an address has been re-allocated. The state-space is less well-equipped to handle the approximation on thread ids. When abstracting thread ids, we could keep a set of abstract threads paired with the store. But, it is natural to define the forthcoming concrete and abstract transitions when the set of threads becomes a map. Since every thread has a distinct thread id, we can model the set of threads in each state as a partial map from a thread id to a context:

$$\mathit{Threads} \equiv \mathit{TID} \rightharpoonup \mathit{Context}.$$

It is straightforward to update the concurrent transition relation when it calls out to the sequential transition relation:

$$\frac{(c, \sigma) \to (c', \sigma')}{(T[t \mapsto c], \sigma) \Rightarrow (T[t \mapsto c'], \sigma').}$$

2.4 Concurrent transition in the P(CEK*)S machine

We define the concurrent transitions separately from the sequential transitions. For instance, if a context is attempting to `spawn` a thread, the concurrent relation handles it by allocating a new thread id t', and binding it to the new context c'':

$$\begin{aligned}
(T[t \mapsto \overbrace{(\llbracket \texttt{(spawn } e\texttt{)} \rrbracket, \rho, a_\kappa, h)}^{c}], \sigma) \Rightarrow (T[t \mapsto c', t' \mapsto c''], \sigma'),& \\
\text{where } t' &= \mathit{newtid}(c, T[t \mapsto c]) \\
c'' &= (e, \rho, a_{\mathbf{halt}}, h_0) \\
h' &= \mathit{record}(c, h) \\
(v', e', \rho', a'_\kappa) &= \sigma(a_\kappa) \\
a' &= \mathit{alloc}(v', h') \\
\rho'' &= \rho'[v' \mapsto a'] \\
c' &= (e', \rho'', a'_\kappa, h') \\
\sigma' &= \sigma[a' \mapsto t'], \text{ where:}
\end{aligned}$$

– *newtid* : $\mathit{Context} \times \mathit{Threads} \to \mathit{TID}$ allocates a fresh thread id for the newly spawned thread.
– *record* : $\mathit{Context} \times \mathit{Hist} \to \mathit{Hist}$ is responsible for updating the history of execution with this context.
– *alloc* : $\mathsf{Var} \times \mathit{Hist} \to \mathit{Addr}$ allocates a fresh address for the supplied variable.

The abstract counterparts to these functions determine the degree of approximation in the analysis, and consequently, the trade-off between speed and precision.

When a thread halts, its thread id is treated as an address, and its return value is stored there:

$$(T[t \mapsto ([\![æ]\!], \rho, a_{\mathbf{halt}}, h)], \sigma) \Rightarrow (T, \sigma[t \mapsto \mathcal{E}(æ, \rho, \sigma)]).$$

This convention, of using thread ids as addresses, makes it easy to model thread joins, since they can check to see if that address has value waiting or not:

$$\frac{\sigma(\mathcal{E}(æ, \rho, \sigma)) = d}{(T[t \mapsto \underbrace{([\![\texttt{(join}\ æ\texttt{)}]\!], \rho, a_\kappa, h)}_{c}], \sigma) \Rightarrow (T[t \mapsto (e, \rho', a'_\kappa, h')], \sigma'),}$$

$$\begin{aligned} \text{where } \kappa &= \sigma(a_\kappa) \\ (v, e, \rho, a'_\kappa) &= \kappa \\ \rho' &= \rho[v \mapsto a''] \\ h' &= \mathit{record}(c, h) \\ a'' &= \mathit{alloc}(v, h') \\ \sigma' &= \sigma[a'' \mapsto d]. \end{aligned}$$

3 A systematic abstract interpretation of P(CEK*)S

Using the techniques outlined in our recent work on systematically constructing abstract interpretations from abstract machines [23], we can directly convert the P(CEK*)S machine into an abstract interpretation of itself. In the concrete state-space, there are four points at which we must inflict abstraction: over basic values (like numbers), over histories, over addresses and over thread ids.

The abstraction over histories determines the context-sensitivity of the analysis on a per-thread basis. The abstraction over addresses determines polyvariance. The abstraction over thread ids maps concrete threads into abstract threads, which determines to what extent the analysis can distinguish dynamically created threads from one another; it directly impacts MHP analysis.

The abstract state-space (Figure 2) mirrors the concrete state-space in structure. We assume the natural point-wise, element-wise and member-wise lifting of a partial order ($\sqsubseteq$) over all of the sets within the state-space. Besides the restriction of histories, addresses and thread ids to finite sets, it is also worth

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \widehat{\mathit{Threads}} \times \widehat{\mathit{Store}} \\
\hat{T} \in \widehat{\mathit{Threads}} &= \widehat{\mathit{TID}} \to \mathcal{P}(\widehat{\mathit{Context}}) \\
\hat{c} \in \widehat{\mathit{Context}} &= \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Addr}} \times \widehat{\mathit{Hist}} \\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &= \mathsf{Var} \times \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Addr}} + \{\mathbf{halt}\} \\
\hat{h} \in \widehat{\mathit{Hist}} & \text{ contains bounded, finite program histories} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \to \hat{D} \\
\hat{d} \in \hat{D} &= \mathcal{P}(\widehat{\mathit{Value}}) \\
\widehat{\mathit{val}} \in \widehat{\mathit{Value}} &= \widehat{\mathit{Clo}} + \mathit{Bool} + \widehat{\mathit{Num}} + \widehat{\mathit{Kont}} + \widehat{\mathit{TID}} + \widehat{\mathit{Addr}} \\
\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}} \\
\hat{a} \in \widehat{\mathit{Addr}} & \text{ is a finite set of abstract addresses} \\
\hat{t} \in \widehat{\mathit{TID}} & \text{ is a finite set of abstract thread ids}
\end{aligned}
$$

Fig. 2. Abstract state-space for a systematic abstraction of the P(CEK*)S machine.

pointing out that the range of both $\widehat{\mathit{Threads}}$ and $\widehat{\mathit{Store}}$ are *power* sets. This promotion occurs because, during the course of an analysis, re-allocating the same thread id or address is all but inevitable. To maintain soundness, the analysis must be able to store multiple thread contexts in the same abstract thread id, and multiple values at the same address in the store.

The structural abstraction map α on the state-space (Figure 3) utilizes a family of abstraction maps over the sets within the state-space. With the abstraction and the abstract state-space fixed, the abstract transition relation reduces to a matter of calculation [4]. The relation $(\leadsto)$ describes the concurrent abstract transition, while the relation $(\multimap)$ describes the sequential abstract transition:

$$
\begin{aligned}
(\leadsto) &\subseteq \hat{\Sigma} \times \hat{\Sigma} \\
(\multimap) &\subseteq (\widehat{\mathit{Context}} \times \widehat{\mathit{Store}}) \times (\widehat{\mathit{Context}} \times \widehat{\mathit{Store}})
\end{aligned}
$$

When the context in focus is sequential, the sequential relation takes over:

$$
\frac{(\hat{c}, \hat{\sigma}) \multimap (\hat{c}', \hat{\sigma}')}{(\hat{T}[\hat{t} \mapsto \{\hat{c}\} \cup \hat{C}], \hat{\sigma}) \leadsto (\hat{T} \sqcup [\hat{t} \mapsto \{\hat{c}'\}], \hat{\sigma}')}
$$

There is a critical change over the concrete rule in this abstract rule: thanks to the join operation, the abstract context remains associated with the abstract thread id even after its transition has been considered. In the next section, we will examine the application of singleton abstraction to thread ids to allow the "strong update" of abstract threads ids across transition. (For programs whose

$$
\begin{aligned}
\alpha_{\Sigma}(T,\sigma) &= (\alpha(T),\alpha(\sigma))\\
\alpha_{\mathit{Threads}}(T) &= \lambda\hat{t}.\bigsqcup_{\alpha(t)=\hat{t}} \alpha(T(t))\\
\alpha_{\mathit{Context}}(e,\rho,\kappa,h) &= \{(e,\alpha(\rho),\alpha(\kappa),\alpha(h))\}\\
\alpha_{\mathit{Env}}(\rho) &= \lambda v.\alpha(\rho(v))\\
\alpha_{\mathit{Kont}}(v,e,\rho,a) &= (v,e,\alpha(\rho),\alpha(a))\\
\alpha_{\mathit{Kont}}(\mathbf{halt}) &= \mathbf{halt}\\
\alpha_{\mathit{Store}}(\sigma) &= \lambda\hat{a}.\bigsqcup_{\alpha(a)=\hat{a}} \alpha(\sigma(a))\\
\alpha_{D}(\mathit{val}) &= \{\alpha(\mathit{val})\}\\
\alpha_{\mathit{Clo}}(\mathit{lam},\rho) &= (\mathit{lam},\alpha(\rho))\\
\alpha_{\mathit{Bool}}(b) &= b\\
\alpha_{\mathit{Hist}}(h) &\text{ is defined by context-sensitivity}\\
\alpha_{\mathit{TID}}(t) &\text{ is defined by thread-sensitivity}\\
\alpha_{\mathit{Addr}}(a) &\text{ is defined by polyvariance.}
\end{aligned}
$$

Fig. 3. A structural abstraction map.

maximum number of threads is statically bounded by a known constant, this allows for precise MHP analysis.)

3.1 Running the analysis

Given a program e, the injection function $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{\mathit{State}}$ creates the initial abstract machine state:

$$\hat{\mathcal{I}}(e) = \left(\left[\hat{t}_0 \mapsto \left\{(e,[],\hat{a}_{\mathbf{halt}},\hat{h}_0)\right\}\right], [\hat{a}_{\mathbf{halt}} \mapsto \{\mathbf{halt}\}]\right),$$

where $\hat{h}_0$ is a blank abstract history and $\hat{a}_{\mathbf{halt}}$ is the distinguished abstract address of the **halt** continuation. The analysis of a program e is the finite set of states reachable from the initial state:

$$\hat{\mathcal{R}}(e) = \left\{\hat{\varsigma} : \hat{\mathcal{I}}(e) \leadsto^{*} \hat{\varsigma}\right\}.$$

3.2 A fixed point interpretation

If one prefers a traditional, fixed-point abstract interpretation, we can imagine the intermediate state of the analysis itself as a set of currently reachable abstract machine states:

$$\hat{\xi} \in \hat{\Xi} = \mathcal{P}(\hat{\Sigma}).$$

A global transfer function $\hat{f} : \hat{\Xi} \to \hat{\Xi}$ evolves this set:

$$\hat{f}(\hat{\xi}) = \left\{\hat{\mathcal{I}}(e)\right\} \cup \left\{\hat{\varsigma}' : \hat{\varsigma} \in \hat{\xi} \text{ and } \hat{\varsigma} \rightsquigarrow \hat{\varsigma}'\right\}.$$

The solution of the analysis is the least fixed point: $\mathrm{lfp}(\hat{f})$.

3.3 Termination

The dependence structure of the abstract state-space is a directed acyclic graph starting from the set $\hat{\Sigma}$ at the root. Because all of the leaves of this graph (*e.g.*, lambda terms, abstract numbers, abstract addresses) are finite for any given program, the state-space itself must also be finite. Consequently, there are no infinitely ascending chains in the lattice $\hat{\Xi}$. By Kleene's fixed point theorem, there must exist a least natural n such that $\mathrm{lfp}(\hat{f}) = \hat{f}^n(\emptyset)$.

3.4 Concurrent abstract transitions

Guided by the structural abstraction, we can convert the concrete concurrent transitions for the P(CEK*)S machine into concurrent abstract transitions. For instance, if an abstract context is attempting to `spawn` a thread, the concurrent relation handles it by allocating a new thread id $\hat{t}'$, and binding it to the new context $\hat{c}''$:

$$\begin{aligned}
(\hat{T}[\hat{t} \mapsto \{\overbrace{(\llbracket\texttt{(spawn } e\texttt{)}\rrbracket, \hat{\rho}, \hat{a}_{\hat{\kappa}}, \hat{h})}^{\hat{c}}\} \cup \hat{C}], \hat{\sigma}) &\rightsquigarrow (\hat{T} \sqcup [\hat{t} \mapsto \{\hat{c}'\}, \hat{t}' \mapsto \{\hat{c}''\}], \sigma'), \\
\text{where } \hat{t}' &= \widehat{newtid}(\hat{c}, \hat{T}[\hat{t} \mapsto \hat{C} \cup \{\hat{c}\}]) \\
\hat{c}'' &= (e, \hat{\rho}, \hat{a}_{\mathbf{halt}}, \hat{h}_0) \\
\hat{h}' &= \widehat{record}(\hat{c}, \hat{h}) \\
(v', e', \hat{\rho}', \hat{a}'_{\hat{\kappa}}) &\in \hat{\sigma}(\hat{a}_{\hat{\kappa}}) \\
\hat{a}' &= \widehat{alloc}(v', \hat{h}') \\
\hat{\rho}'' &= \hat{\rho}'[v' \mapsto \hat{a}'] \\
\hat{c}' &= (e', \hat{\rho}'', \hat{a}'_{\hat{\kappa}}, \hat{h}') \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}' \mapsto \{\hat{t}'\}], \text{ where:}
\end{aligned}$$

– $\widehat{newtid} : \widehat{Context} \times \widehat{Threads} \to \widehat{TID}$ allocates a thread id for the newly spawned thread.
– $\widehat{record} : \widehat{Context} \times \widehat{Hist} \to \widehat{Hist}$ is responsible for updating the (bounded) history of execution with this context.
– $\widehat{alloc} : \mathsf{Var} \times \widehat{Hist} \to \widehat{Addr}$ allocates an address for the supplied variable.

These functions determine the degree of approximation in the analysis, and consequently, the trade-off between speed and precision.

When a thread halts, its abstract thread id is treated as an address, and its return value is stored there:

$$\frac{\hat{T}' = \hat{T} \sqcup [\hat{t} \mapsto \{([\![æ]\!], \hat{\rho}, \hat{a}_{\mathbf{halt}}, \hat{h})\}]}{(\hat{T}', \sigma) \leadsto (\hat{T}', \hat{\sigma} \sqcup [\hat{t} \mapsto \hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma})])}, \text{ where}$$

the atomic evaluation function $\hat{\mathcal{E}} : \mathsf{AExp} \times \widehat{Env} \times \widehat{Store} \to \hat{D}$ maps an atomic expression to a value in the context of an environment and a store:

$$\hat{\mathcal{E}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v))$$
$$\hat{\mathcal{E}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}.$$

It is worth asking whether it is sound in just this case to remove the context from the threads (making the subsequent threads T instead of T'). It is sound, but it seems to require a (slightly) more complicated staggered-state bisimulation to prove it: the concrete counterpart to this state may take several steps to eliminate all of its halting contexts.

Thanks to the convention to use thread ids as addresses holding the return value of thread, it easy to model thread joins, since they can check to see if that address has a value waiting or not:

$$\frac{\hat{a} \in \hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma}) \qquad \hat{d} = \hat{\sigma}(\hat{a})}{(\hat{T} \sqcup [\hat{t} \mapsto \{\underbrace{([\![\texttt{(join } æ\texttt{)}]\!], \hat{\rho}, \hat{a}_{\hat{\kappa}}, \hat{h})}_{\hat{c}}\}], \hat{\sigma}) \leadsto (\hat{T} \sqcup [\hat{t} \mapsto \{(e, \hat{\rho}', \hat{a}'_{\hat{\kappa}}, \hat{h}'), \hat{c}\}], \hat{\sigma}'),}$$

$$\begin{aligned} \text{where } \hat{\kappa} &\in \hat{\sigma}(\hat{a}_{\hat{\kappa}}) \\ (v, e, \hat{\rho}, \hat{a}'_{\hat{\kappa}}) &= \hat{\kappa} \\ \hat{\rho}' &= \hat{\rho}[v \mapsto \hat{a}''] \\ \hat{h}' &= \widehat{record}(\hat{c}, \hat{h}) \\ \hat{a}'' &= \widehat{alloc}(v, \hat{h}') \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}'' \mapsto \hat{d}]. \end{aligned}$$

3.5 Soundness

Compared to a standard proof of soundness for a small-step abstract interpretation, the proof of soundness requires only slightly more attention in a concurrent setting. The key lemma in the inductive proof of simulation states that when a concrete state ς abstracts to an abstract state $\hat{\varsigma}$, if the concrete state can transition to ς', then the abstract state $\hat{\varsigma}$ must be able to transition to some other abstract state $\hat{\varsigma}'$ such that ς' abstracts to $\hat{\varsigma}'$:

Theorem 1. *If:*

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma} \textit{ and } \varsigma \Rightarrow \varsigma',$$

then there must exist a state $\hat{\varsigma}'$ such that:

$$\alpha(\varsigma') \sqsubseteq \hat{\varsigma}' \text{ and } \hat{\varsigma} \rightsquigarrow \hat{\varsigma}'.$$

Proof. The proof is follows the case-wise structure of proofs like those in Might and Shivers [14]. There is an additional preliminary step: first choose a thread id modified across transition, and then perform case-wise analysis on how it could have been modified.

3.6 Extracting flow information

The core question in flow analysis is, "Can the value *val* flow to the expression æ?" To answer it, assume that $\hat{\xi}$ is the set of all reachable abstract states. We must check every state within this set, and every environment $\hat{\rho}$ within that state. If $\hat{\sigma}$ is the store in that state, then *val* may flow to æ if the value $\alpha(val)$ is represented in the set $\hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma})$. Formally, we can construct a flows-to relation, $\mathit{FlowsTo} \subseteq \mathit{Value} \times \mathsf{AExpr}$, for a program e:

$$\mathit{FlowsTo}(val, æ) \text{ iff there exist } (\hat{T}, \hat{\sigma}) \in \hat{\mathcal{R}}(e) \text{ and } \hat{t} \in \mathit{dom}(\hat{T}) \text{ such that}$$
$$(e, \hat{\rho}, \hat{\kappa}) \in \hat{T}(\hat{t}) \text{ and } \{\alpha(val)\} \sqsubseteq \hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma}).$$

3.7 Extracting MHP information

In MHP analysis, we are concerned with whether two expressions e' and e'' may be evaluated concurrently with one another in program e. It is straightforward to decide this using the set of reachable states computed by the abstract interpretation. If, in any reachable state, there exist two distinct contexts at the relevant expressions, then their evaluation may happen in parallel with one another. Formally, the $\mathit{MHP} \subseteq \mathsf{Exp} \times \mathsf{Exp}$ relation with respect to program e is:

$$\mathit{MHP}(e', e'') \text{ iff there exist } (\hat{T}, \hat{\sigma}) \in \hat{\mathcal{R}}(e) \text{ and } \hat{t}', \hat{t}'' \in \mathit{dom}(\hat{T}) \text{ such that}$$
$$(e', \_, \_, \_) \in \hat{T}(\hat{t}') \text{ and } (e'', \_, \_, \_) \in \hat{T}(\hat{t}'').$$

4 MHP: Making strong transitions with singleton threads

In the previous section, we constructed a systematic abstraction of the P(CEK*)S machine. While it serves as a sound and capable flow analysis, its precision as an MHP analysis is just above useless. Contexts associated with each abstract thread id grow monotonically during the course of the analysis. Eventually, it will seem as though every context may happen in parallel with every other context. By comparing the concrete and abstract semantics, the cause of the imprecision becomes clear: where the concrete semantics *replaces* the context at a given thread id, the abstract semantics *joins*.

Unfortunately, we cannot simply discard the join. A given abstract thread id could be representing multiple concrete thread ids. Discarding a thread id would then discard possible interleavings, and it could even introduce unsoundness.

Yet, it is plainly the case that *many* programs have a boundable number of threads that are co-live. Thread creation is considered expensive, and thread pools created during program initialization are a popular mechanism for circumventing the problem. To exploit this design pattern, we can make thread ids eligible for "strong update" across transition. In shape analysis, strong update refers to the ability to treat an abstract address as the representative of a single concrete address when assigning to that address. That is, by tracking the cardinality of the abstraction of each thread id, we can determine when it is sound to replace functional join with functional update *on threads themselves.*

The necessary machinery is straightforward, adapted directly from the shape analysis literature [1,2,11,10,14,19] ; we attach to each state a cardinality counter $\hat{\mu}$ that tracks how many times an abstract thread id has been allocated (but not precisely beyond once):

$$\hat{\varsigma} \in \hat{\Sigma} = \widehat{Threads} \times \widehat{Store} \times \widehat{TCount}$$
$$\hat{\mu} \in \widehat{TCount} = \widehat{TID} \to \{0, 1, \infty\} .$$

When the count of an abstract thread id is exactly one, we know for certain that there exists at most one concrete counterpart. Consequently, it is safe to perform a "strong transition." Consider the case where the context in focus for the concurrent transition is sequential; in the case where the count is exactly one, the abstract context gets replaced on transition:

$$\frac{(\hat{c}, \hat{\sigma}) \multimap (\hat{c}', \hat{\sigma}') \qquad \hat{\mu}(\hat{t}) = 1}{(\hat{T}[\hat{t} \mapsto \{\hat{c}\} \uplus \hat{C}], \hat{\sigma}, \hat{\mu}) \leadsto (\hat{T}[\hat{t} \mapsto \{\hat{c}'\} \cup \hat{C}], \hat{\sigma}', \hat{\mu}).}$$

It is straightforward to modify the existing concurrent transition rules to exploit information available in the cardinality counter. At the beginning of the analysis, all abstract thread ids have a count of zero. Upon spawning a thread, the analysis increments the result of the function $\widehat{newtid}$. When a thread whose abstract thread id has a count of one halts, its count is reset to zero.

4.1 Strategies for abstract thread id allocation

Just as the introduction of an allocation function for addresses provides the ability to tune polyvariance, the $\widehat{newtid}$ function provides the ability to tune precision. The optimal strategy for allocating this scarce pool of abstract thread ids depends upon the design patterns in use.

One could, for instance, allocate abstract thread ids according to calling context, *e.g.*, the abstract thread id is the last k call sites. This strategy would work well for the implementation of futures, where futures from the same context are often not co-live with themselves.

The context-based strategy, however, is not a reasonable strategy for a thread-pool design pattern. All of the spawns will occur at the same expression in the same loop, and therefore, in the same context. Consequently, there will be no discrimination between threads. If the number of threads in the pool is known *a priori* to be n, then the right strategy for this pattern is to create n abstract thread ids, and to allocate a new one for each iteration of the thread-pool-spawning loop. On the other hand, if the number of threads is set dynamically, no amount of abstract thread ids will be able to discriminate between possible interleavings effectively, in this case a reasonable choice for precision would be to have one abstract thread per thread pool.

4.2 Advantages for MHP analysis

With the cardinality counter, it is possible to test whether an expression may be evaluated in parallel with itself. If, for every state, every abstract thread id which maps to a context containing that expression has a count of one, and no other context contains that expression, then that expression must never be evaluated in parallel with itself. Otherwise, parallel evaluation is possible.

5 Flow analysis of concurrent higher-order programs

If the concern is a sound flow analysis, but not MHP analysis, then we can perform an abstract interpretation of our abstract interpretation that efficiently collapses all possible interleavings and paths, even as it retains limited (reachability-based) flow-sensitivity. This second abstraction map $\alpha' : \hat{\Xi} \to \hat{\Sigma}$ operates on the system-space of the fixed-point interpretation:

$$\alpha'(\hat{\xi}) = \bigsqcup_{\hat{\varsigma} \in \hat{\xi}} \hat{\varsigma}.$$

The new transfer function, $\hat{f}' : \hat{\Sigma} \to \hat{\Sigma}$, monotonically accumulates all of the visited states into a single state:

$$\hat{f}'(\hat{\varsigma}) = \hat{\varsigma} \sqcup \bigsqcup_{\hat{\varsigma} \Rightarrow \hat{\varsigma}'} \hat{\varsigma}'.$$

5.1 Complexity

This second abstraction simplifies the calculation of an upper bound on computational complexity. The structure of the set $\hat{\Sigma}$ is a pair of maps into sets:

$$\hat{\Sigma} = \left(\widehat{TID} \to \mathcal{P}(\widehat{Context})\right) \times \left(\widehat{Addr} \to \mathcal{P}(\widehat{Value})\right).$$

Each of these maps is, in effect, a table of bit vectors: the first with abstract thread ids on one axis and contexts on the other; and the second with abstract addresses on one axis and values on the other. The analysis monotonically flips bits on each pass. Thus, the maximum number of passes—the tallest ascending chain in the lattice $\hat{\Sigma}$—is:

$$|\widehat{TID}| \times |\widehat{Context}| + |\widehat{Addr}| \times |\widehat{Value}|.$$

Thus, the complexity of the analysis is determined by context-sensitivity, as with classical sequential flow analysis. For a standard monovariant analysis, the complexity is polynomial [17]. For a context-sensitive analysis with shared environments, the complexity is exponential [22]. For a context-sensitive analysis with flat environments, the complexity is again polynomial [15].

6 Related work

This work traces its ancestry to Cousot and Cousot's work on abstract interpretation [3,4]. We could easily extend the fixed-point formulation with the implicit concretization function to arrive at an instance of traditional abstract interpretation. It is also a direct descendant of the line of work investigating control-flow in higher-programs that began with Jones [12] and Shivers [20,21].

The literature on static analysis of concurrency and higher-orderness is not empty, but it is spare. Much of it focuses on the special case of the analysis of futures. The work most notable and related to our own is that of Navabi and Jagannathan [16]. It takes Flanagan and Felleisen's notion of safe futures [7,8], and develops a dynamic and static analysis that can prevent a continuation from modifying a resource that one of its concurrent futures may modify. What makes this work most related to our own is that it is sound even in the presence of exceptions, which are, in essence, an upward-restricted form of continuations. Their work and our own own interact synergistically, since their safety analysis focuses on removing the parallel inefficiencies of safe futures; our flow analysis can remove the sequential inefficiencies of futures through the elimination of run-time type-checks. Yahav's work is the earliest to apply shape-analytic techniques to the analysis of concurrency [24].

It has taken substantial effort to bring the static analysis of higher-order programs to heel; to recite a few of the major challenges:

1. First-class functions from dynamically created closures over lambda terms create recursive dependencies between control- and data-flow; k-CFA co-analyzes control and data to factor and order these dependencies [21].
2. Environment-bearing closures over lambda terms impart fundamental intractabilities unto context-sensitive analysis [22]—intractabilities that were only recently side-stepped via flattened abstract environments [15].
3. The functional emphasis on recursion over iteration made achieving high precision difficult (or hopeless) without abstract garbage collection to recycle tail-call-bound parameters and continuations [14].
4. When closures keep multiple bindings to the same variable live, precise reasoning about side effects to these bindings requires the adaptation of shape-analytic techniques [11,13].
5. Precise reasoning about first-class continuations (and kin such as exceptions) required a harmful conversion to continuation-passing style until the advent of small-step abstraction interpretations for the pointer-refined CESK machine [23].

We see this work as another milestone on the path to robust static analysis of full-featured higher-order programs.

7 Limitations and future work

Since the shape of the store and the values within were not the primary focus of this work, it utilized a blunt abstraction. A compelling next step of this work would generalize the abstraction of the store relationally, so as to capture relations between the values at specific *addresses*. The key challenge in such an extension is the need to handle relations between abstract addresses which may represent *multiple* concrete addresses. Relations which universally quantify over the concrete constituents of an abstract address are a promising approach.

Acknowledgements This material is based upon work supported by the NSF under Grant No. 1035658. The second author was supported by the National Science Foundation under Grant No. 0937060 to the Computing Research Association for the CIFellow Project. We also acknowledge support from the Park City Research Institute for Computer Science.

References

1. Balakrishnan, G., and Reps, T. Recency-Abstraction for Heap-Allocated storage. In *SAS '06: Static Analysis Symposium* (Berlin, Heidelberg, 2006), K. Yi, Ed., vol. 4134 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 221–239.
2. Chase, D. R., Wegman, M., and Zadeck, F. K. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), PLDI '90, ACM, pp. 296–310.
3. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* (New York, NY, USA, 1977), ACM Press, pp. 238–252.
4. Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1979), POPL '79, ACM Press, pp. 269–282.
5. Feeley, M. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University, Apr. 1993.
6. Felleisen, M., and Friedman, D. P. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts* (Aug. 1986).
7. Flanagan, C., and Felleisen, M. The semantics of future and its use in program optimization. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), ACM, pp. 209–220.
8. Flanagan, C., and Felleisen, M. The semantics of future and an application. *Journal of Functional Programming 9*, 01 (1999), 1–31.

9. Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, June 1993), ACM, pp. 237–247.
10. Hudak, P. A semantic model of reference counting and its abstraction. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1986), ACM, pp. 351–363.
11. Jagannathan, S., Thiemann, P., Weeks, S., and Wright, A. Single and loving it: must-alias analysis for higher-order languages. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 329–341.
12. Jones, N. D. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming* (London, UK, 1981), Springer-Verlag, pp. 114–128.
13. Might, M. Shape analysis in the absence of pointers and structure. In *VMCAI 2010: International Conference on Verification, Model-Checking and Abstract Interpretation* (Madrid, Spain, Jan. 2010), pp. 263–278.
14. Might, M., and Shivers, O. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming 18*, Special Double Issue 5-6 (2008), 821–864.
15. Might, M., Smaragdakis, Y., and Van Horn, D. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), ACM, pp. 305–315.
16. Navabi, A., and Jagannathan, S. Exceptionally safe futures. In *Coordination Models and Languages*, J. Field and V. Vasconcelos, Eds., vol. 5521 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009, ch. 3, pp. 47–65.
17. Palsberg, J. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems 17*, 1 (Jan. 1995), 47–62.
18. Queinnec, C. Continuations and web servers. *Higher-Order and Symbolic Computation 17*, 4 (Dec. 2004), 277–295.
19. Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems 24*, 3 (May 2002), 217–298.
20. Shivers, O. Control flow analysis in Scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988), ACM, pp. 164–174.
21. Shivers, O. G. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.
22. Van Horn, D., and Mairson, H. G. Deciding kCFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2008), ACM, pp. 275–282.
23. Van Horn, D., and Might, M. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2010), ACM, pp. 51–62.
24. Yahav, E. Verifying safety properties of concurrent java programs using 3-valued logic. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2001), POPL '01, ACM Press, pp. 27–40.