Automated and modular refinement reasoning for concurrent programs

```
var x:int;

yielding procedure p()
  requires x >= 5;
  ensures  x >= 8;
{
  yield x >= 5;  x := x + 1;
  yield x >= 6;  x := x + 1;
  yield x >= 7;  x := x + 1;
}

procedure q() modifies x; { x := x + 3; }
```

Figure 1: Program 1

Abstract

We present a verifier for concurrent programs based on automated and modular refinement reasoning.

1. Introduction

2. Overview

3. Overview

We present an overview of the CIVL language through a sequence of examples. Figure 1 shows Program 1 containing a procedure p executing concurrently with another procedure q. An execution of a CIVL program is non-preemptive; a thread explicitly yields control to the scheduler via the `yield` statement following which execution continues on a nondeterministically chosen thread. The `yield` statement has a local assertion φ attached to it. The yielding thread must establish φ when it yields and the execution of other threads must preserve φ; these two requirements are usually known as *sequential correctness* and *non-interference*, respectively. To check these requirements, the CIVL verifier creates verification conditions, whose number is at most quadratic in the number of yield statements in the program. For example, in Program 1 each yield predicate in p must be checked against the action `x := x + 3` in q.

CIVL requires that a procedure that may potentially execute a yield statement during its execution must be annotated as `yielding`. This annotation is checked in a manner similar to the checking of modifies clauses; if a procedure is labeled as `yielding` so must all of its callers. A procedure marked as `yielding` is exempt from providing a modifies clause; the presence of `yielding` allows the caller to conclude that any global variable could have changed potentially as a result of modification by a concurrently-executing thread. A procedure not labeled as `yielding` is called atomic; such a procedure must supply a modifies clause as usual.

From quadratic to linear verification conditions. Figure 2 shows Program 2, a variation of Program 1 in which the procedure

```
var x:int;

yielding procedure yield_x(n:int)
  requires x >= n;
  ensures  x >= n;
{
  yield x >= n;
}

yielding procedure p()
  requires x >= 5;
  ensures  x >= 8;
{
  call yield_x(5);  x := x + 1;
  call yield_x(6);  x := x + 1;
  call yield_x(7);  x := x + 1;
}
```

Figure 2: Program 2

```
yielding procedure yield_x(n: int)
  requires x >= n;
  ensures  x >= n;
{
  yield x >= n;
}

yielding procedure p()
  requires x >= 5;
  ensures  x >= 8;
{
  call yield_x(x);  x := x + 1;
  call yield_x(x);  x := x + 1;
  call yield_x(x);  x := x + 1;
}
```

Figure 3: Program 3

`yield_x` contains a single yield statement and p calls `yield_x` instead of yielding directly. If the calls to `yield_x` are inlined in Program 2, then we will get Program 1. Both Program 1 and 2 are verifiable in CIVL but the cost of verifying Program 2 is less because it has fewer yield statements. In fact, if it is possible to capture all interference in a concurrent program in a single yield predicate, then the trick in Program 2 can be used to verify the program with a linear number of verification conditions.

Encoding rely-guarantee specifications. Figure 3 shows Program 3, yet another variation of Programs 1 and 2 which shows how to encode a rely-guarantee-style [8] (two-state invariant) proof using CIVL's one-state yield statements. The standard rely-guarantee specification to prove the assertions in p is that the environment

```
var x:int, y:int, z:int;

stable yielding procedure incr_x()
  ensures  x >= old(x) + 1;
{
  yield x >= old(x);
  x := x + 1;
  yield x >= old(x) + 1;
}

stable yielding procedure yield_y()
  ensures  y >= old(y);
{
  yield y >= old(y);
}

stable yielding procedure yield_z()
  ensures  z >= old(z);
{
  yield z >= old(z);
}

yielding procedure p()
  requires x == 3 && y == 5 && z == 7;
{
  call incr_x() | yield_y() | yield_z();
  assert x >= 4 && y >= 5 && z >= 7;
}
```

Figure 4: Program 4

```
type Tid;
var linear alloc:[Tid]bool;
const nil: Tid;
procedure Allocate() returns (linear tid: Tid);
  modifies alloc;
  ensures tid != nil;

var a:[Tid]int;

yielding procedure main()
{
  var linear tid: Tid;
  while (true) {
    call tid := Allocate();
    async call P(tid);
    yield true;
  }
}

yielding procedure P(linear tid: Tid)
  requires tid != nil;
  ensures a[tid] == old(a)[tid] + 1;
{
  var t: int;
  t := a[tid];
  yield t == a[tid];
  a[tid] := t + 1;
}
```

Figure 5: Program 5

of p may only increase x. We can encode this in CIVL by first exploiting the trick in Program 2 to factor out the yield statement in a separate procedure and then passing the current value of x as a parameter to yield_x. In fact, our implementation of CIVL requires even less work; the value of x upon entering yield_x is available in the postcondition using the syntax old(x), allowing us to write yield_x without any parameter as follows:

```
yielding procedure yield_x()
  ensures  x >= old(x);
{
  yield x >= old(x);
}
```

Parallel calls. Program 4 in Figure 4 illustrates the parallel call feature of CIVL, based on the standard Owicki-Gries rules for parallel composition of threads. The statement `call incr_x() | yield_y() | yield_z()` in p creates three threads executing incr_x, yield_y, and yield_z respectively, yields control to the scheduler, and blocks until all three threads have terminated. For a procedure to be invoked in a parallel call, it must be annotated as stable. This annotation indicates to the CIVL verifier that the precondition and postcondition of the procedure must be stable against interference. This requirement ensures that it is safe to assume the precondition in the callee and the postcondition in the caller.

The threads created by p for yield_y and yield_z are not doing any interesting computation; their only purpose is to make available to their parent the conjunction of their respective postconditions (following Owicki-Gries rules for parallel composition). In this example, the postconditions of yield_y and yield_z preserve information about variables y and z that would otherwise be lost during the call to incr_x, whose postcondition only supplies information about x even though its yield statements potentially cause all global variables, including y and z, to change. This example shows that CIVL allows modular proof structuring by factoring out yield assertions into a collection of procedures; the declaration of incr_x can focus on changes to x, without having to explicitly preserve invariants about all other variables in the program.

Linear variables. Program 5 in Figure 6 introduces linear variables, a feature of CIVL that is useful for encoding disjointness among values contained in different variables. This example uses this feature for encoding the concept of an identifier that is unique to each thread. Program 5 contains a shared global array a indexed by an uninterpreted type Tid representing the set of thread identifiers. A collection of threads are executing procedure P concurrently. The identifier of the thread executing P is passed in as the parameter tid. A thread with identifier tid owns a[tid] and can increment it without danger of interference. The yield assertion `t == a[tid]` in P indicates this expectation, yet it is not possible to prove it unless the reasoning engine knows that the value of tid in one thread is distinct its value in a different thread.

Instead of building a notion of thread identifiers into CIVL, we provide a more primitive and general notion of linear variables. The CIVL type system ensures that values contained in linear variables cannot be duplicated. Consequently, the parameter tid of distinct concurrent calls to P are known to be distinct; the CIVL verifier exploits this invariant while checking for non-interference.

Program 5 also shows the mechanisms of allocation of thread identifiers, based on the use of global variable alloc, the constant nil, and the procedure Allocate. Section **??** describes values and linear variables like nil and alloc in more detail.

Refinement.

```
var Color:int;

procedure {:yields} SetColGrayIfWhite({:cnst "tid"} tid:int)
ensures {:atomic} [if (Color == WHITE())
                        Color := GRAY();]
{
  call cNoLock:= GetColorNoLock();
  call YieldColorOnlyGetsDarker();
  if (cNoLock == WHITE()) {
       call L_SetColorToGrayIfWhite(tid);
  }
}

procedure {:yields} YieldColorOnlyGetsDarker()
  ensures Color >= old(Color);

procedure {:yields} L_SetColGrayIfWhite({:cnst "tid"} tid:int)
ensures {:atomic} [if (Color == WHITE())
                        Color := GRAY();]
{
  call AcquireLock(tid);
  call cLock := GetColorLocked(tid);
  if (cLock == WHITE()) {
    call SetColorLocked(tid, GRAY());
  }
  call ReleaseLock(tid);
}
```

Figure 6: Program 6

4. A simple programming language

5. Verification

5.1 Type checking

5.2 Safety

Non-interference. Let $Yields$ be the union of the following sets:

- $\{(\phi, \lambda) \mid \mathit{yield}\ \phi, \lambda$ appears in $Prog\}$.
- $\{(\phi, \lambda) \mid P \in ProcName \wedge ps(P) = (\phi, M, \psi, s) \wedge ls(P) = (\lambda, \lambda')\}$.
- $\{(\psi, \lambda') \mid P \in ProcName \wedge ps(P) = (\phi, M, \psi, s) \wedge ls(P) = (\lambda, \lambda')\}$.

Let $Ablocks$ be the set of atomic blocks in the program except those inside the bodies of procedures in $dom(RS)$. Let $FV \subseteq VarName \setminus Vars$ be a set of fresh variables and Λ be a one-one substitution function from $ThreadLocals \cup Locals$ to FV. Let $\Lambda(\phi)$ represent the result of applying Λ to the expression ϕ. We define

$$D(\lambda_y, \lambda_a) = disjoint(\{x \mid x \in \lambda_y\} \cup \{x \mid x \in \lambda_a\})$$

For each predicate $(\phi, \lambda_y) \in Yields$ and for each atomic block $\mathit{ablock}\ \{e, \lambda_a\}\ s \in Ablocks$, we prove the following judgment:

$$\{\Lambda(\phi) \wedge e \wedge D(\Lambda(\lambda_y \setminus Globals), \lambda_a)\} s \{\Lambda(\phi)\}$$

For each predicate $\phi \in Yields$ and for each $A \in range(RS)$ such that $as(A) = (\rho, \alpha, m)$ and $ls(A) = (\lambda_a, \lambda'_a)$, we prove the following to be unsatisfiable:

$$(\Lambda(\phi) \wedge \rho \wedge D(\Lambda(\lambda_y \setminus Globals), \lambda_a)) \circ \alpha \circ \neg\Lambda(\phi)$$

Program Syntax

$$\begin{array}{rcl}
g & \in & Globals \subseteq VarName \\
tl & \in & ThreadLocals \subseteq VarName \\
l & \in & Locals \subseteq VarName \\
x, y & \in & Vars = Globals \cup ThreadLocals \cup Locals \\
v & \in & Value \\
\sigma & \in & Store = Vars \to Value \\
G & \in & StoreGlobals = Globals \to Value \\
TL & \in & StoreThreadLocals = ThreadLocals \to Value \\
L & \in & StoreLocals = Locals \to Value \\
e, \phi, \psi, \rho & \in & StateExpr = 2^{Store} \\
\alpha, \beta & \in & TransExpr = 2^{(Store, Store)} \\
le & \in & LocalStateExpr = 2^{StoreLocals} \\
P & \in & ProcName \\
A & \in & ActionName \\
m & \in & Mover = \{B, R, L, N\} \\
as & \in & ActionName \to (StateExpr, TransExpr, Mover) \\
ps & \in & ProcName \to (StateExpr, 2^{ThreadLocals}, StateExpr, Stmt) \\
RS & \in & ProcName \rightharpoonup ActionName \\
\lambda & \in & LinearVars = 2^{Globals \cup ThreadLocals} \\
ls & \in & (ActionName \cup ProcName) \to (LinearVars, LinearVars) \\
a & \in & InsideABlock ::= \mathrm{a}^+ \mid \mathrm{a}^- \\
r & \in & InsideRefinement ::= \mathrm{r}^+ \mid \mathrm{r}^-
\end{array}$$

$$\begin{array}{rcl}
s \in Stmt & ::= & \mathit{skip} \mid \mathit{assert}\ le \mid \mathit{yield}\ e, \lambda \mid \\
 & & \mathit{call}\ A \mid \mathit{call}\ P \mid \mathit{async}\ P \mid \\
 & & \mathit{ablock}\ \{e, \lambda\}\ s \mid s;\ s \mid \\
 & & \mathit{if}\ le\ \mathit{then}\ s\ \mathit{else}\ s \mid \\
 & & \mathit{while}\ \{e, \alpha\}\ le\ \mathit{do}\ s \\
SS \in StmtStack & ::= & s \mid (L, SS) \mid SS; s \\
T \in Thread & ::= & (TL, (L, SS)) \\
Prog \in Program & ::= & PC[G][TL][L][s] \\
 & & \\
SC \in StmtCtxt & ::= & []_{Stmt} \mid SC; s \\
SSC \in StmtStackCtxt & ::= & ([]_{Locals}, SC) \mid (L, SSC) \mid SSC; s \\
TC \in ThreadCtxt & ::= & ([]_{ThreadLocals}, ([]_{Locals}, SC)) \mid \\
 & & ([]_{ThreadLocals}, (L, SSC)) \\
YT \in YieldingThread & ::= & TC[TL][L][\mathit{yield}\ e, \lambda] \\
PC \in ProgCtxt & ::= & (ps, as, ls, []_{Globals}, \overrightarrow{YT} \cdot TC \cdot \overrightarrow{YT})
\end{array}$$

Figure 7: Syntax

$$\frac{\begin{array}{c} PC[G][TL][L][s] = (\_, as, \_, \_, \_) \\ as \vdash (G \cdot TL \cdot L, s) \longrightarrow (G' \cdot TL' \cdot L', s') \end{array}}{PC[G][TL][L][s] \longrightarrow PC[G'][TL'][L'][s']} \quad (\textsc{Program-Step})$$

$$\frac{PC[G][TL][L][s] = (\_, as, \_, \_, \_) \qquad as \vdash (G \cdot TL \cdot L, s)\ \text{fails}}{PC[G][TL][L][s])\ \text{fails}} \quad (\textsc{Program-Fail})$$

$$\frac{\begin{array}{c} ps(P) = (\phi, M, \psi, s) \\ ls(P) = (\lambda, \lambda') \qquad T' = (TL, (L, \mathit{yield}\ \phi, \lambda; s)) \end{array}}{(ps, as, ls, G, TC[TL][L][\mathit{async}\ P]) \longrightarrow (ps, as, ls, G, TC[TL][L][\mathit{skip}] \cdot T')} \quad (\textsc{Async})$$

$$\frac{}{(ps, as, ls, G, \overrightarrow{YT} \cdot (TL, (L, \mathit{skip})) \cdot \overrightarrow{YT}') \longrightarrow (ps, as, ls, G, \overrightarrow{YT} \cdot \overrightarrow{YT}')} \quad (\textsc{Thread-End})$$

$$\frac{\begin{array}{c} ps(P) = (\phi, M, \psi, s) \\ ls(P) = (\lambda, \lambda') \qquad SS = \mathit{yield}\ \phi, \lambda; s; \mathit{yield}\ \psi, \lambda' \end{array}}{PC[G][TL][L][\mathit{call}\ P] \longrightarrow PC[G][TL][L][(L, SS)]} \quad (\textsc{Call})$$

$$\frac{}{PC[G][TL][L][(L', \mathit{skip})] \longrightarrow PC[G][TL][L][\mathit{skip}]} \quad (\textsc{Return})$$

Figure 8: Operational semantics for program

$$\frac{\sigma = G{\cdot}TL{\cdot}L \qquad L \in le}{as \vdash (\sigma, assert\ le) \longrightarrow (\sigma, skip)} \quad (\textsc{Assert-True})$$

$$\frac{\sigma = G{\cdot}TL{\cdot}L \qquad L \notin le}{as \vdash (\sigma, assert\ le) \text{ fails}} \quad (\textsc{Assert-False})$$

$$\frac{as(A) = (\rho, \alpha, m) \qquad (\sigma, \sigma') \in \alpha}{as \vdash (\sigma, call\ A) \longrightarrow (\sigma', skip)} \quad (\textsc{Atomic})$$

$$\frac{}{as \vdash (\sigma, yield\ e, \lambda) \longrightarrow (\sigma, skip)} \quad (\textsc{Yield})$$

$$\frac{}{as \vdash (\sigma, ablock\ \{e, \lambda\}\ s\ ) \longrightarrow (\sigma, s)} \quad (\textsc{AtomicBlock})$$

$$\frac{}{as \vdash (\sigma, skip; s) \longrightarrow (\sigma, s)} \quad (\textsc{Seq})$$

$$\frac{\sigma = G{\cdot}TL{\cdot}L \qquad L \in le}{as \vdash (\sigma, if\ le\ then\ s_1\ else\ s_2\ ) \longrightarrow (\sigma, s_1)} \quad (\textsc{If-True})$$

$$\frac{\sigma = G{\cdot}TL{\cdot}L \qquad L \notin le}{as \vdash (\sigma, if\ le\ then\ s_1\ else\ s_2\ ) \longrightarrow (\sigma, s_2)} \quad (\textsc{If-False})$$

$$\frac{\sigma = G{\cdot}TL{\cdot}L \qquad L \notin le}{as \vdash (\sigma, while\ \{e, \alpha\}\ le\ do\ s\ ) \longrightarrow (\sigma, skip)} \quad (\textsc{While-False})$$

$$\frac{\sigma = G{\cdot}TL{\cdot}L \qquad L \in le}{as \vdash (\sigma, while\ \{e, \alpha\}\ le\ do\ s\ ) \longrightarrow (\sigma, s; while\ \{e, \alpha\}\ le\ do\ s\ )} \quad (\textsc{While-True})$$

Figure 9: Operational semantics for statement

5.3 Refinement

5.4 Yield sufficiency

Commutativity. Let $FV_1, FV_2 \subseteq VarName \setminus Vars$ be two sets of disjoint fresh variables. Let Λ_1 and Λ_2 be one-one substitution functions from $ThreadLocals \cup Locals$ to FV_1 and FV_2 respectively. For all $A_1, A_2 \in ActionName$ such that $as(A_1) = (\rho_1, \alpha_1, m_1)$, $as(A_2) = (\rho_2, \alpha_2, m_2)$, $ls(A_1) = (\lambda_1, \lambda_1')$, and$ls(A_2) = (\lambda_2, \lambda_2')$, if $m_1 \in \{B, R\}$ or $m_2 \in \{B, L\}$ then prove the following valid:

$$\begin{aligned}&(\Lambda_1(\rho_1) \wedge \Lambda_2(\rho_2) \wedge D(\Lambda_1(\lambda_1 \setminus Globals), \Lambda_2(\lambda_2)))\\ &\circ(\Lambda_1(\alpha_1) \wedge Same(FV_2)) \circ (\Lambda_2(\alpha_2) \wedge Same(FV_1))\\ &\Rightarrow (\Lambda_2(\alpha_2) \wedge Same(FV_1)) \circ (\Lambda_1(\alpha_1) \wedge Same(FV_2))\end{aligned}$$

For all $A_1, A_2 \in ActionName$ such that $as(A_1) = (\rho_1, \alpha_1, m_1)$ and $as(A_2) = (\rho_2, \alpha_2, m_2)$, if $m_1 \in \{B, R\}$ then prove the following unsatisfiable:

$$\Lambda_1(\rho_1) \circ (\Lambda_2(\rho_2) \circ D(\Lambda_1(\lambda_1 \setminus Globals), \Lambda_2(\lambda_2)) \circ \Lambda_2(\alpha_2)) \circ \neg\Lambda_1(\rho_1)$$

For all $A_1, A_2 \in ActionName$ such that $as(A_1) = (\rho_1, \alpha_1, m_1)$ and $as(A_2) = (\rho_2, \alpha_2, m_2)$, if $m_1 \in \{B, L\}$ then prove the following unsatisfiable:

$$\neg\Lambda_1(\rho_1) \circ (\Lambda_2(\rho_2) \circ D(\Lambda_1(\lambda_1 \setminus Globals), \Lambda_2(\lambda_2)) \circ \Lambda_2(\alpha_2)) \circ \Lambda_1(\rho_1)$$

For all $A \in ActionName$ such that $as(A) = (\rho, \alpha, m)$ and $m \in \{B, L\}$, prove the following valid:

$$\forall \sigma \in \rho.\ \exists \sigma'.\ (\sigma, \sigma') \in \alpha$$

5.5 Program refinement

5.6 Soundness

Theorem 1. *Let $Prog$ and $Prog'$ be two programs. Let $RS \in ProcName \rightharpoonup ActionName$ be a partial function from procedure names to action names. Let $AS \in 2^{ActionName}$ be a set of action names. Suppose the following conditions are satisfied:*

$$\frac{}{\lambda; a \vdash skip : \lambda} \quad (\textsc{Skip}) \qquad \frac{}{\lambda; \mathrm{a}^- \vdash assert\ le : \lambda} \quad (\textsc{Assert})$$

$$\frac{\lambda_y \subseteq \lambda}{\lambda; \mathrm{a}^- \vdash yield\ e, \lambda_y : \lambda} \quad (\textsc{Yield}) \qquad \frac{ls(A) = (\lambda, \lambda')}{\lambda; \mathrm{a}^+ \vdash call\ A : \lambda'} \quad (\textsc{Atomic})$$

$$\frac{ls(P) = (\lambda, \lambda')}{\lambda; \mathrm{a}^- \vdash call\ P : \lambda'} \quad (\textsc{Proc}) \qquad \frac{\begin{array}{c}\lambda_G \subseteq Globals\\ \lambda \cup \lambda_P \cup \lambda_P' \subseteq ThreadLocals \qquad ls(P) = ((\lambda_G, \lambda_P), (\lambda_G, \lambda\end{array}}{\lambda_G, \lambda, \lambda_P; \mathrm{a}^- \vdash async\ P : \lambda_G, \lambda}$$

$$\frac{\lambda; \mathrm{a}^+ \vdash s : \lambda' \qquad \lambda_a \subseteq \lambda}{\lambda; \mathrm{a}^- \vdash ablock\ \{e, \lambda_a\}\ s\ : \lambda'} \quad (\textsc{Ablock})$$

$$\frac{\lambda; a \vdash SS : \lambda'}{\lambda; a \vdash (L, SS) : \lambda'} \quad (\textsc{StackFrame})$$

$$\frac{\lambda; a \vdash SS : \lambda' \qquad \lambda'; a \vdash s : \lambda''}{\lambda; a \vdash SS; s : \lambda''} \quad (\textsc{Seq})$$

$$\frac{\lambda; a \vdash s_1 : \lambda' \qquad \lambda; a \vdash s_2 : \lambda'}{\lambda; a \vdash if\ le\ then\ s_1\ else\ s_2\ : \lambda'} \quad (\textsc{Ite})$$

$$\frac{\lambda; a \vdash s : \lambda}{\lambda; a \vdash while\ \{e, \alpha\}\ le\ do\ s\ : \lambda} \quad (\textsc{While})$$

$$\frac{ls(P) = (\lambda, \lambda') \qquad ps(P) = (\phi, M, \psi, s) \qquad \lambda; \mathrm{a}^- \vdash s : \lambda'}{\vdash P} \quad (\textsc{Procedure})$$

$$\frac{\begin{array}{c}ls(A) = (\lambda, \lambda') \qquad \lambda \cap Globals = \lambda' \cap Globals \qquad as(A) = (\rho, \alpha, m)\\ \forall (\sigma, \sigma') \in \alpha. disjoint(\{\sigma(x) \mid x \in \lambda\}) \wedge disjoint(\{\sigma'(x) \mid x \in \lambda'\})\\ \forall (\sigma, \sigma') \in \alpha. \bigcup\{\sigma'(x) \mid x \in \lambda'\} \subseteq \bigcup\{\sigma(x) \mid x \in \lambda\}\end{array}}{\vdash A} \quad (\textsc{Action})$$

$$\frac{T = (TL, (L, SS)) \qquad \lambda; \mathrm{a}^- \vdash SS : \lambda'}{\lambda \vdash T} \quad (\textsc{Thread})$$

$$\frac{\begin{array}{c}\forall P \in ProcName. \vdash P \qquad \forall A \in ActionName. \vdash A\\ \lambda_G \subseteq Globals \qquad \forall 1 \leq i \leq n.(\lambda_G, \lambda_i \vdash T_i)\\ \forall 1 \leq i \leq n.(\lambda_i \subseteq ThreadLocals) \qquad \forall 1 \leq i \leq n.(T_i = (TL_i, \ldots))\\ disjoint(\{G(x) \mid x \in \lambda_G\} \cup \{TL_i(x) \mid 1 \leq i \leq n, x \in \lambda_i\})\end{array}}{\vdash (ps, as, ls, G, T_1 \ldots T_n)} \quad (\textsc{Program})$$

Figure 10: Type checking rules

1. *$\vdash Prog$ and $\vdash Prog'$ and $RS; AS \vdash Prog \rightsquigarrow Prog'$.*
2. *All finite executions of $Prog'$ are safe.*
3. *All infinite executions of $Prog'$ are responsive.*
4. *The program $Prog$ is commutativity-safe.*
5. *The program $Prog$ is interference-free.*
6. *$RS \vdash_p Prog$ and $RS \vdash_r Prog$ and $RS \vdash_y Prog$.*

Then all finite executions of $Prog$ are safe.

5.7 Responsiveness

$$\frac{\begin{array}{c}ps; as; r; M \vdash_p \{e \wedge le\}s\{e\}\\ e \wedge le \Rightarrow f \geq 0 \qquad s \preceq (old(e \wedge le) \Rightarrow old(f) > f)\end{array}}{ps; as; r; M \vdash_p \{e\}while\ \{e, \alpha, f\}\ le\ do\ s\ \{e \wedge \neg le\}} \quad (\textsc{While})$$

6. Implementation

7. A concurrent garbage collector

We have chosen to demonstrate the proposed verification methodology and tool on a realistic modern concurrent garbage collector. To this end, we designed a garbage collector that extends the concurrent collector of Dijkstra et. al. [4]. The goal in this design is to build a collector that is on one hand easy to verify and on the

$$\frac{}{ps; as; r; \{\} \vdash_p \{\phi\} \mathit{skip} \{\phi\}}\ (\textsc{Skip})$$

$$\frac{}{ps; as; \mathrm{r}^-; \{\} \vdash_p \{\phi\} \mathit{assert}\ le \{\phi\}}\ (\textsc{Assert1})$$

$$\frac{}{ps; as; \mathrm{r}^+; \{\} \vdash_p \{\phi \wedge le\} \mathit{assert}\ le \{\phi\}}\ (\textsc{Assert2})$$

$$\frac{}{ps; as; r; \{\} \vdash_p \{e\} \mathit{yield}\ e, \lambda \{e\}}\ (\textsc{Yield})$$

$$\frac{\begin{array}{c} as(A) = (\rho, \alpha, m) \\ M \subseteq \mathit{ThreadLocals} \qquad \alpha \Rightarrow \mathit{Havoc}(\mathit{Globals} \cup M \cup \mathit{Locals}) \\ \phi \Rightarrow \rho \qquad \mathit{Unsat}(\phi \circ \alpha \circ \neg\psi) \end{array}}{ps; as; r; M \vdash_p \{\phi\} \mathit{call}\ A \{\psi\}}\ (\textsc{Atomic})$$

$$\frac{ps(P) = (\phi, M, \psi, s) \qquad P \notin \mathit{dom}(RS)}{ps; as; r; M \vdash_p \{\phi\} \mathit{call}\ P \{\psi\}}\ (\textsc{Proc1})$$

$$\frac{\begin{array}{c} ps(P) = (\phi, M, \psi, s) \qquad P \in \mathit{dom}(RS) \\ ps; as; r; M \vdash_p \{\phi\} \mathit{call}\ RS(P) \{\psi\} \end{array}}{ps; as; r; M \vdash_p \{\phi\} \mathit{call}\ P \{\psi\}}\ (\textsc{Proc2})$$

$$\frac{ps(P) = (\phi, M, \psi, s)}{ps; as; r; \{\} \vdash_p \{\rho \wedge \phi\} \mathit{async}\ P \{\rho\}}\ (\textsc{Async})$$

$$\frac{ps; as; r; M \vdash_p \{\phi_1 \wedge e\} s \{\phi_2\}}{ps; as; r; M \vdash_p \{\phi_1 \wedge e\} \mathit{ablock}\ \{e, \lambda\}\ s\ \{\phi_2\}}\ (\textsc{Ablock})$$

$$\frac{ps; as; r; M_1 \vdash_p \{\phi_1\} SS \{\phi_2\} \qquad ps; as; r; M_2 \vdash_p \{\phi_2\} s \{\phi_3\}}{ps; as; r; M_1 \cup M_2 \vdash_p \{\phi_1\} SS; s \{\phi_3\}}\ (\textsc{Seq})$$

$$\frac{\begin{array}{c} ps; as; r; M_1 \vdash_p \{e \wedge \phi_1\} s_1 \{\phi_2\} \\ ps; as; r; M_2 \vdash_p \{\neg e \wedge \phi_1\} s_2 \{\phi_2\} \end{array}}{ps; as; r; M_1 \cup M_2 \vdash_p \{\phi_1\} \mathit{if}\ le\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ \{\phi_2\}}\ (\textsc{Ite})$$

$$\frac{ps; as; r; M \vdash_p \{e \wedge le\} s \{e\}}{ps; as; r; M \vdash_p \{e\} \mathit{while}\ \{e, \alpha\}\ le\ \mathit{do}\ s\ \{e \wedge \neg le\}}\ (\textsc{While})$$

$$\frac{\phi \Rightarrow \phi' \qquad ps; as; r; M \vdash_p \{\phi'\} SS \{\psi'\} \qquad \psi' \Rightarrow \psi}{ps; as; r; M \vdash_p \{\phi\} SS \{\psi\}}\ (\textsc{Weaken})$$

$$\frac{ps; as; r; M \vdash_p \{\phi\} SS \{\psi\} \qquad \mathit{Acc}(\rho) \cap M = \{\}}{ps; as; r; M \vdash_p \{\rho \wedge \phi\} SS \{\rho \wedge \psi\}}\ (\textsc{Frame})$$

$$\frac{\begin{array}{c} ps(P) = (\phi, M, \psi, s) \qquad r = \mathrm{r}^+ \iff P \in \mathit{dom}(RS) \\ ps; as; r; M' \vdash_p \{\phi\} s \{\psi\} \qquad M' \subseteq M \end{array}}{ps; as; RS \vdash_p P}\ (\textsc{Procedure})$$

$$\frac{\begin{array}{c} ps; as; r; M \vdash_p \{\phi'\} SS \{\psi\} \qquad (\mathit{Acc}(\phi) \cup \mathit{Acc}(\psi)) \cap \mathit{Locals} = \{\} \\ \forall G, \mathit{TL}, L.\ G \cdot \mathit{TL} \cdot L \in \phi \Rightarrow G \cdot \mathit{TL} \cdot L \in \phi' \end{array}}{ps; as; r; M \vdash_p \{\phi\} (L', SS) \{\psi\}}\ (\textsc{StackFrame})$$

$$\frac{\begin{array}{c} T = (\mathit{TL}, (L, SS)) \\ G \cdot \mathit{TL} \cdot L \in \phi \qquad ps; as; \mathrm{r}^-; M \vdash_p \{\phi\} SS \{\mathit{true}\} \end{array}}{ps; as; G \vdash_p T}\ (\textsc{Thread})$$

$$\frac{\forall P \in \mathit{ProcName}.\ ps; as; RS \vdash_p P \qquad \forall 1 \le i \le n.\ ps; as; G \vdash_p T_i}{RS \vdash_p (ps, as, ls, G, T_1 \dots T_n)}\ (\textsc{Program})$$

Figure 11: Sequential rules for partial correctness

$$\frac{}{RS; as; P \vdash_r \mathit{skip} : \epsilon}\ (\textsc{Skip}) \qquad \frac{}{RS; as; P \vdash_r \mathit{assert}\ le : \epsilon}\ (\textsc{Assert})$$

$$\frac{}{RS; as; P \vdash_r \mathit{yield}\ e, \lambda : \epsilon}\ (\textsc{Yield})$$

$$\frac{\begin{array}{c} P' \in \mathit{dom}(RS) \qquad as(RS(P')) = (\rho', \alpha', m) \\ \rho' \circ \alpha' \Rightarrow \mathit{Havoc}(\{\}) \end{array}}{RS; as; P \vdash_r \mathit{call}\ P' : \epsilon}\ (\textsc{Call-Loop})$$

$$\frac{\begin{array}{c} P' \in \mathit{dom}(RS) \\ as(RS(P)) = (\rho, \alpha, m) \qquad as(RS(P')) = (\rho', \alpha', m) \\ \mathit{OldLocals} = \mathit{old}(\mathit{Locals}) \qquad \rho' \circ \alpha' \Rightarrow \exists \mathit{OldLocals}, \mathit{Locals}.\ \alpha \end{array}}{RS; as; P \vdash_r \mathit{call}\ P' : N}\ (\textsc{Call-Action})$$

$$\frac{\begin{array}{c} P' \in \mathit{dom}(RS) \\ as(RS(P')) = (\rho', \alpha', m) \qquad \rho' \circ \alpha' \Rightarrow \mathit{Havoc}(\{\}) \end{array}}{RS; as; P \vdash_r \mathit{async}\ P' : \epsilon}\ (\textsc{Async})$$

$$\frac{as \vdash s \preceq \mathit{old}(e) \Rightarrow \mathit{Havoc}(L)}{RS; as; P \vdash_r \mathit{ablock}\ \{e, \lambda\}\ s\ : \epsilon}\ (\textsc{Ablock-Loop})$$

$$\frac{\begin{array}{c} as(RS(P)) = (\rho, \alpha, m) \qquad \mathit{OldLocals} = \mathit{old}(\mathit{Locals}) \\ as \vdash s \preceq \mathit{old}(e) \Rightarrow \exists \mathit{OldLocals}, \mathit{Locals}.\ \alpha \end{array}}{RS; as; P \vdash_r \mathit{ablock}\ \{e, \lambda\}\ s\ : N}\ (\textsc{Ablock-Action})$$

$$\frac{RS; as; P \vdash_r s_1 : re_1 \qquad RS; as; P \vdash_r s_2 : re_2}{RS; as; P \vdash_r s_1; s_2 : re_1 \cdot re_2}\ (\textsc{Seq})$$

$$\frac{RS; as; P \vdash_r s_1 : re_1 \qquad RS; as; P \vdash_r s_2 : re_2}{RS; as; P \vdash_r \mathit{if}\ le\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ : re_1 + re_2}\ (\textsc{Ite})$$

$$\frac{RS; as; P \vdash_r s : re}{RS; as; P \vdash_r \mathit{while}\ \{e, \alpha\}\ le\ \mathit{do}\ s\ : re^*}\ (\textsc{While})$$

$$\frac{ps(P) = (\phi, M, \psi, s) \qquad \forall P \in \mathit{dom}(RS).\ RS; as; P \vdash_r s : \{N\}}{RS \vdash_r (ps, as, ls, G, T_1 \dots T_n)}\ (\textsc{Program})$$

Figure 12: Refinement rules

$$\frac{}{as \vdash \mathit{skip} \preceq \mathit{Havoc}(\{\})}\ (\textsc{Skip}) \qquad \frac{}{as \vdash \mathit{assert}\ le \preceq \mathit{Havoc}(\{\})}\ (\textsc{Assert})$$

$$\frac{}{as \vdash \mathit{yield}\ e, \lambda \preceq \mathit{false}}\ (\textsc{Yield}) \qquad \frac{as(A) = (\rho, \alpha, m)}{as \vdash \mathit{call}\ A \preceq \alpha}\ (\textsc{Atomic})$$

$$\frac{}{as \vdash \mathit{call}\ P \preceq \mathit{false}}\ (\textsc{Call}) \qquad \frac{}{as \vdash \mathit{async}\ P \preceq \mathit{Havoc}(\{\})}\ (\textsc{Async})$$

$$\frac{s \preceq \alpha}{as \vdash \mathit{ablock}\ \{e, \lambda\}\ s\ \preceq \alpha}\ (\textsc{Ablock})$$

$$\frac{as \vdash s_1 \preceq \alpha_1 \qquad as \vdash s_2 \preceq \alpha_2}{as \vdash s_1; s_2 \preceq \alpha_1 \circ \alpha_2}\ (\textsc{Seq})$$

$$\frac{as \vdash s_1 \preceq \alpha_1 \qquad as \vdash s_2 \preceq \alpha_2}{as \vdash \mathit{if}\ le\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ \preceq (le \circ \alpha_1) \vee (\neg le \circ \alpha_2)}\ (\textsc{Ite})$$

$$\frac{as \vdash s \preceq \beta \qquad \neg le \circ \mathit{Havoc}(\{\}) \Rightarrow \alpha \qquad \beta \circ \alpha \Rightarrow \alpha}{as \vdash \mathit{while}\ \{e, \alpha\}\ le\ \mathit{do}\ s\ \preceq e \circ \alpha \circ \neg le}\ (\textsc{While})$$

$$\frac{as \vdash s \preceq \alpha \qquad \alpha \Rightarrow \alpha'}{as \vdash s \preceq \alpha'}\ (\textsc{Weaken})$$

Figure 13: Abstracting statements by actions

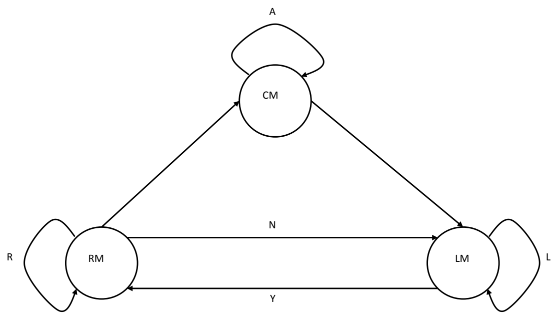

Figure 14: Specification for yield sufficiency

$$\frac{}{RS; as \vdash_y skip : (x, x)}\ (\textsc{Skip}) \qquad \frac{}{RS; as \vdash_y assert\ le : (x, x)}\ (\textsc{Assert})$$

$$\frac{}{RS; as \vdash_y yield\ e, \lambda : (x, RM)}\ (\textsc{Yield})$$

$$\frac{P \in dom(RS) \qquad as(RS(P)) = (\rho, \alpha, B)}{RS; as \vdash_y call\ P : (x, x)}\ (\textsc{CallBothMover})$$

$$\frac{P \in dom(RS) \qquad as(RS(P)) = (\rho, \alpha, R)}{RS; as \vdash_y call\ P : (RM, RM)}\ (\textsc{CallRightMover})$$

$$\frac{P \in dom(RS) \qquad as(RS(P)) = (\rho, \alpha, L)}{RS; as \vdash_y call\ P : (x, LM)}\ (\textsc{CallLeftMover})$$

$$\frac{P \in dom(RS) \qquad as(RS(P)) = (\rho, \alpha, N)}{RS; as \vdash_y call\ P : (RM, LM)}\ (\textsc{CallNonMover})$$

$$\frac{P \notin dom(RS)}{RS; as \vdash_y call\ P : (x, RM)}\ (\textsc{CallYield})$$

$$\frac{}{RS; as \vdash_y async\ P : (x, LM)}\ (\textsc{Async})$$

$$\frac{x \in \{RM, CM\}}{RS; as \vdash_y ablock\ \{e, \lambda\}\ s\ : (x, CM)}\ (\textsc{Ablock})$$

$$\frac{RS; as \vdash_y SS : (x, y) \qquad RS; as \vdash_y s : (y, z)}{RS; as \vdash_y SS; s : (x, z)}\ (\textsc{Seq})$$

$$\frac{RS; as \vdash_y s_1 : (x, y) \qquad RS; as \vdash_y s_2 : (x, y)}{RS; as \vdash_y if\ le\ then\ s_1\ else\ s_2\ : (x, y)}\ (\textsc{Ite})$$

$$\frac{RS; as \vdash_y s : (x, x)}{RS; as \vdash_y while\ \{e, \alpha\}\ le\ do\ s\ : (x, x)}\ (\textsc{While})$$

$$\frac{ps(P) = (\phi, M, \psi, s) \qquad RS; as \vdash_y s : (x, y)}{RS; as \vdash_y P}\ (\textsc{Procedure})$$

$$\frac{RS; as \vdash_y SS : (x, y)}{RS; as \vdash_y (L, SS) : (x, y)}\ (\textsc{StackFrame})$$

$$\frac{T = (TL, (L, SS)) \qquad RS; as \vdash_y (L, SS) : (x, y)}{RS; as \vdash_y T}\ (\textsc{Thread})$$

$$\frac{\begin{array}{c}\forall P \in ProcName \setminus dom(RS).\ RS; as \vdash_y P \\ \forall 1 \leq i \leq n.\ RS; as \vdash_y T_i\end{array}}{RS \vdash_y (ps, as, ls, G, T_1 \dots T_n)}\ (\textsc{Program})$$

Figure 15: Yield sufficiency rules

$$\frac{}{RS; AS \vdash skip \rightsquigarrow skip}\ (\textsc{Skip}) \qquad \frac{}{RS; AS \vdash assert\ le \rightsquigarrow assert\ le}\ (\textsc{Assert})$$

$$\frac{}{RS; AS \vdash yield\ e, \lambda \rightsquigarrow yield\ e, \lambda}\ (\textsc{Yield})$$

$$\frac{A \notin AS}{RS; AS \vdash call\ A \rightsquigarrow call\ A}\ (\textsc{Atomic})$$

$$\frac{RS; AS \vdash s \rightsquigarrow s'}{RS; AS \vdash ablock\ \{e, \lambda\}\ s\ \rightsquigarrow s'}\ (\textsc{Ablock-Elim})$$

$$\frac{RS; AS \vdash s \rightsquigarrow s'}{RS; AS \vdash s \rightsquigarrow ablock\ \{e, \lambda\}\ s'}\ (\textsc{Ablock-Intro})$$

$$\frac{P \in dom(RS)}{RS; AS \vdash call\ P \rightsquigarrow call\ RS(P)}\ (\textsc{Proc1})$$

$$\frac{P \notin dom(RS)}{RS; AS \vdash call\ P \rightsquigarrow call\ P}\ (\textsc{Proc2})$$

$$\frac{}{RS; AS \vdash async\ P \rightsquigarrow async\ P}\ (\textsc{Async})$$

$$\frac{RS; AS \vdash SS \rightsquigarrow SS'}{RS; AS \vdash (L, SS) \rightsquigarrow (L, SS')}\ (\textsc{StackFrame})$$

$$\frac{RS; AS \vdash SS \rightsquigarrow SS' \qquad RS; AS \vdash s \rightsquigarrow s'}{RS; AS \vdash SS; s \rightsquigarrow SS'; s'}\ (\textsc{Seq})$$

$$\frac{RS; AS \vdash s_1 \rightsquigarrow s_1' \qquad RS; AS \vdash s_2 \rightsquigarrow s_2'}{RS; AS \vdash if\ le\ then\ s_1\ else\ s_2\ \rightsquigarrow if\ le\ then\ s_1'\ else\ s_2'}\ (\textsc{Ite})$$

$$\frac{RS; AS \vdash s \rightsquigarrow s'}{RS; AS \vdash while\ \{e, \alpha\}\ le\ do\ s\ \rightsquigarrow while\ \{e, \alpha\}\ le\ do\ s'}\ (\textsc{While})$$

$$\frac{RS; AS \vdash s \rightsquigarrow s'}{RS; AS \vdash (\phi, M, \psi, s) \rightsquigarrow (\phi', M', \psi', s')}\ (\textsc{Procedure})$$

$$\frac{\begin{array}{c}Acc(\rho) \cap Locals = \emptyset \qquad OldLocals = old(Locals) \\ \alpha = ((\exists OldLocals, Locals.\ \alpha) \wedge Same(Locals))\end{array}}{RS; AS \vdash (\rho, \alpha, m) \rightsquigarrow (\rho, \alpha, m)}\ (\textsc{Action})$$

$$\frac{RS; AS \vdash SS \rightsquigarrow SS'}{RS; AS \vdash (TL, (L, SS)) \rightsquigarrow (TL, (L, SS'))}\ (\textsc{Thread})$$

$$\frac{\begin{array}{c}Prog = (ps, as, ls, G, T_1 \dots T_n) \\ Prog' = (ps', as', ls', G, T_1' \dots T_n') \\ \forall 1 \leq i \leq n.\ RS; AS \vdash T_i \rightsquigarrow T_i' \qquad \forall A \notin AS.\ as(A) = as'(A) \\ \forall A \in range(RS).\ RS; AS \vdash as(A) \rightsquigarrow as(A) \\ \forall P \notin dom(RS).\ RS; AS \vdash ps(P) \rightsquigarrow ps'(P) \\ \forall P \in dom(RS).\ ls(P) = ls(RS(P))\end{array}}{RS; AS \vdash Prog \rightsquigarrow Prog'}\ (\textsc{Program})$$

Figure 16: Program refinement

other hand highly performant in practice. Most modern concurrent collectors are snapshot-oriented [1, 5–7], and as such may require complex claims on snapshot times and reachability. Other popular concurrent collectors and in particular the *mostly concurrent* garbage collector [2, 3, 9] consist of different complex phases and complex interaction between objects that reside on specific *cards*. We preferred a collector whose invariants are simple and hold continuously as much as possible throughout the execution. This means that a program execution can move from one thread to another and then to the garbage collector while all relevant invariants continue to hold at all times.

Dijkstra's collector seems to be a good candidate, but it cannot be considered a modern or performant collector. On the positive side, its write-barrier maintains simple invariants continuously. However, this collector becomes incorrect in the presence of more than one program thread (mutator), which is unacceptable for use with modern multicore platforms. Furthermore, it requires that the write-barrier will run with both heap pointers and root pointers, i.e., on any modification of the runtime stacks and the registers. This requirement implies bad performance, and modern concurrent garbage collectors avoid such a requirement so that they may obtain good performance.

We therefore extended and modified Dijkstra's collector to make it work with parallel programs and also not require applying any write-barrier on root modifications. As far as we know, the obtained garbage collector algorithm has not been previously proposes or implemented. Let us shortly recall Dijkstra's collector, and then explain how we modified it to eliminate its shortcomings.

The analysis of Dijkstra's collector employs a tri-color abstraction to describe the trace of the objects reachable from the roots. Objects are said to be *white* if the collector has not seen them yet during the trace. This means that unreachable objects (that are never encountered during the trace) remain white throughout the trace. Objects that the collector encounters become gray and remain gray until the collector scans their children. Once all the children of an object are noted (meaning that none of them are white), the object becomes black. The collector works by choosing a gray node, *shading* all it's children, and making it black. The shading operation grays a node if it is white, and does noting otherwise. The trace initiates by making all objects white and then shading all objects reachable from the roots. The trace terminates when there are no more gray objects in the heap. Termination is guaranteed because objects can only get darker. Correctness is guaranteed using the invariant that there cannot be a black to white pointer during the trace. At the end of the trace, objects pointed by the roots must be black and since black objects can only point to black objects (there are no gray objects at the end and no black to white pointers), then the entire set of objects reachable from the roots must be black.

In the presence of concurrent program operations, the no-black-to-white invariant does not hold, because the program may simply redirect a pointer of a black object to point to a white object. Therefore, a coordination between the program and the concurrent collector is required and this coordination takes place in the form of a *write-barrier*. A write-barrier is a piece of code that executes with each pointer update, and Dijkstra's write-barrier lets the program shades the new target of a pointer modification. When a pointer field p is set to reference an object B, the write barrier starts by assigning the address of B to p and then it shades B. A major complication comes from the fact that the pointer change and the shading operations do not occur as a single atomic unit, but there is a point in between the two in which the no black-to-white pointers invariant is not preserved. Dijkstra et. al. show that the algorithm is still correct, but only a single program thread. If there are two concurrent program threads, the proof not only fails, but the algorithm actually becomes incorrect.

We start with achieving multithreading and later we will also explain how to get rid of the need to use a write-barrier when modifying the root pointers. We modify Dijkstra's write-barrier by reversing the order of the two actions. First, shade the target object B and only then execute the assignment of B's address to the pointer p. As noted by Doligez et. al. [5, 6] this requires a *handshake* between all program threads and the collector before a new collection initiates. A handshake is initiated by the collector by raising a collector flag that the program threads check occasionally. When a program thread discovers that a handshake is required to start a collection, it raises its own thread flag to indicate that it has noted the handshake flag and that it is not in the middle of a write operation. When the collector finds that all program threads have responded the handshake is done and a collection may start. We use the same handshaking mechanism, but unlike [5, 6], we shade the target of the pointer B. (Most previous concurrent collectors shaded the object that lost a pointer in the operation in order to obtain a snapshot-like tracing behavior). The shading of the target pointer does not make a difference for correctness if we use the write-barrier when modifying root pointers, but next we would like to explain how we avoid the need of using the write-barrier on pointer modifications of the roots.

If we do not use a write-barrier with the roots, it is possible that the roots point to objects that have not been traced, i.e., white objects. This foils the correctness, because it is no longer the case that all objects referenced by the roots are black (and all their descendants are black as well). However, we still know that there are no gray objects at the end of the trace and that there are never black-to-white pointers throughout the trace. So if we could get all roots to reference black objects we would obtain a correct tracing of the heap. To this end, we modify the algorithm to test the roots at the end of the concurrent trace. If all roots point to black objects, then we are done. Otherwise, we continue the collection by shading all objects reachable by the roots and tracing from them again. In a worst-case theoretical scenario we may need to run many root scans and discover more and more white descendants to trace each time. But in practice we usually finish after the first or second traversal, and we seldom need more than that. So we obtain correctness and termination in all scenarios and we obtain good performance in real-world scenarios.

8. Related work

References

[1] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 38(11), pages 269–281, Anaheim, CA, Nov. 2003. ACM Press.

[2] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, Nov. 2005.

[3] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 26(6), pages 157–164, Toronto, Canada, June 1991. ACM Press.

[4] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, Nov. 1978.

[5] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, Jan. 1994. ACM Press.

[6] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, Jan. 1993. ACM Press.

[7] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 35(5), pages 274–284, Vancouver, Canada, June 2000. ACM Press.

[8] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[9] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In C. Chambers and A. L. Hosking, editors, *2nd International Symposium on Memory Management*, ACM SIGPLAN Notices 36(1), pages 143–154, Minneapolis, MN, Oct. 2000. ACM Press.