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 set of yield predicates, preconditions, and postconditions in the program. Let $\mathit{Ablocks}$ be the set of atomic blocks in the program except those inside the bodies of procedures in $dom(rs)$. Let $FV \subseteq \mathit{VarName} \setminus \mathit{Var}$ be a set of fresh variables and Λ be a one-one substitution function from $\mathit{ThreadLocals} \cup \mathit{Locals}$ to FV. Let $\Lambda(\phi)$ represent the result of applying Λ to the expression ϕ. For each predicate $(\phi, \lambda_y) \in \mathit{Yields}$ and for each atomic block $\mathit{ablock}\ \{e, \lambda_a\}\ s\ \in \mathit{Ablocks}$, we prove the following judgment:

$$\{\Lambda(\phi) \wedge e \wedge \mathit{disjoint}(\{\Lambda(x) \mid x \in \lambda_y\} \cup \{x \mid x \in \lambda_a\})\} s \{\Lambda(\phi)\}$$

For each predicate $\phi \in \mathit{Yields}$ and for each $P \in dom(rs)$ such that $as(rs(P)) = (\rho, \alpha, m)$, we prove the following to be unsatisfiable:

$$\Lambda(\phi) \circ \rho \circ \alpha \circ \neg\Lambda(\phi)$$

5.3 Refinement

5.4 Yield sufficiency

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

$$(\Lambda_1(\rho_1) \wedge \Lambda_2(\rho_2)) \circ (\Lambda_1(\alpha_1) \wedge \mathit{Same}(FV_2)) \circ (\Lambda_1(\alpha_2) \wedge \mathit{Same}(FV_1))$$
$$\Rightarrow (\Lambda_1(\alpha_2) \wedge \mathit{Same}(FV_1)) \circ (\Lambda_1(\alpha_1) \wedge \mathit{Same}(FV_2))$$

Program Syntax

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

$$
\begin{array}{rcl}
s \in \mathit{Stmt} & ::= & \mathit{skip} \mid \mathit{assert}\ le \mid \mathit{yield}\ e, \lambda \mid \\
& & \mathit{call}\ A \mid \mathit{call}\ P \mid \mathit{async}\ P \mid \\
& & \mathit{ablock}\ \{e, \lambda\}\ s \mid s;\ s \mid \\
& & \mathit{if}\ le\ \mathit{then}\ s\ \mathit{else}\ s \mid \\
& & \mathit{while}\ \{e, \alpha\}\ le\ \mathit{do}\ s \\
SS \in \mathit{StmtStack} & ::= & s \mid (L, SS) \mid SS; s \\
T \in \mathit{Thread} & ::= & (TL, (L, SS))
\end{array}
$$

$$
\begin{array}{rcl}
SC \in \mathit{StmtCtxt} & ::= & []_{\mathit{Stmt}} \mid SC; s \\
SSC \in \mathit{StmtStackCtxt} & ::= & ([]_{\mathit{Locals}}, SC) \mid (L, SSC) \mid SSC; s \\
TC \in \mathit{ThreadCtxt} & ::= & ([]_{\mathit{ThreadLocals}}, ([]_{\mathit{Locals}}, SC)) \mid \\
& & ([]_{\mathit{ThreadLocals}}, (L, SSC)) \\
YT \in \mathit{YieldingThread} & ::= & TC[TL][L][\mathit{yield}\ e, \lambda] \\
PC \in \mathit{ProgCtxt} & ::= & \overrightarrow{YT} \cdot TC \cdot \overrightarrow{YT}
\end{array}
$$

$$\mathit{Prog} \ \in \ \mathit{Program} = (bs, as, ps, rs, ls, G, \overrightarrow{T})$$

Figure 7: Syntax

$$\frac{(G \cup TL \cup L, s) \longrightarrow (G' \cup TL' \cup L', s')}{(G, PC[TL][L][s]) \longrightarrow (G', PC[TL'][L'][s'])} \quad (\textsc{Program-Step})$$

$$\frac{(G \cup TL \cup L, s)\ \text{fails}}{(G, PC[TL][L][s])\ \text{fails}} \quad (\textsc{Program-Fail})$$

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

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

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

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

Figure 8: Operational semantics for program

$$\frac{\sigma \vdash le \to true}{(\sigma, assert\ le) \longrightarrow (\sigma, skip)} \quad (\text{ASSERT-TRUE})$$

$$\frac{\sigma \vdash le \to false}{(\sigma, assert\ le)\ \text{fails}} \quad (\text{ASSERT-FALSE})$$

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

$$\frac{}{(\sigma, yield\ e, \lambda) \longrightarrow (\sigma, skip)} \quad (\text{YIELD})$$

$$\frac{}{(\sigma, ablock\ \{e, \lambda\}\ s\ ) \longrightarrow (\sigma, s)} \quad (\text{ATOMICBLOCK})$$

$$\frac{}{(\sigma, skip; s) \longrightarrow (\sigma, s)} \quad (\text{SEQ})$$

$$\frac{\sigma \vdash le \to true}{(\sigma, if\ le\ then\ s_1\ else\ s_2\ ) \longrightarrow (\sigma, s_1)} \quad (\text{IF-TRUE})$$

$$\frac{\sigma \vdash le \to false}{(\sigma, if\ le\ then\ s_1\ else\ s_2\ ) \longrightarrow (\sigma, s_2)} \quad (\text{IF-FALSE})$$

$$\frac{\sigma \vdash le \to false}{(\sigma, while\ \{e, \alpha\}\ le\ do\ s\ ) \longrightarrow (\sigma, skip)} \quad (\text{WHILE-FALSE})$$

$$\frac{\sigma \vdash le \to true}{(\sigma, while\ \{e, \alpha\}\ le\ do\ s\ ) \longrightarrow (\sigma, s; while\ \{e, \alpha\}\ le\ do\ s\ )} \quad (\text{WHILE-TRUE})$$

Figure 9: Operational semantics for statement

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:

$$\Lambda_1(\rho_1) \circ (\Lambda_2(\rho_2) \circ \Lambda_2(\alpha_2)) \circ \neg\Lambda_1(\rho_1)$$

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:

$$\neg\Lambda_1(\rho_1) \circ (\Lambda_2(\rho_2) \circ \Lambda_2(\alpha_2)) \circ \Lambda_1(\rho_1)$$

5.5 Program refinement

5.6 Soundness

Theorem 1. *Let Prog and Prog′ be two programs and $AS \subseteq ActionName$ a set of action names such that the following conditions are satisfied:*

1. $\vdash Prog$ *and* $\vdash Prog'$ *and* $AS \vdash Prog \rightsquigarrow Prog'$.
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. $\vdash_p Prog$ *and* $\vdash_r Prog$ *and* $\vdash_y Prog$.

Then all finite executions of Prog are safe.

5.7 Responsiveness

$$\frac{e \wedge le \Rightarrow f \geq 0 \qquad \begin{array}{c}\{e \wedge le\}s\{e\} \\ s \preceq (old(e \wedge le) \Rightarrow old(f) > f)\end{array}}{\{e\}while\ \{e, \alpha, f\}\ le\ do\ s\ \{e \wedge \neg le\}} \quad (\text{WHILE})$$

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.

$$\frac{}{\lambda; a \vdash skip : \lambda} \quad (\text{SKIP}) \qquad \frac{}{\lambda; \mathrm{a}^- \vdash assert\ le : \lambda} \quad (\text{ASSERT})$$

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

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

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

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

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

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

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

$$\frac{ls(P) = (\lambda, \lambda') \qquad \lambda; \mathrm{a}^- \vdash bs(P) : \lambda'}{\vdash P} \quad (\text{PROCEDURE})$$

$$\frac{\begin{array}{c} ls(A) = (\lambda, \lambda') \qquad \lambda \cap Globals = \lambda' \cap Globals \\ as(A) = (\rho, \alpha, m) \qquad m \in \{B, L\} \Longrightarrow \forall\sigma \in \rho.\exists\sigma'.(\sigma, \sigma') \in \alpha \\ A \in range(rs) \Longrightarrow Vars(\rho) \cap Locals = \emptyset \\ OldLocals = old(Locals) \\ A \in range(rs) \Longrightarrow \alpha = ((\exists OldLocals, Locals.\ \alpha) \wedge Same(Locals)) \\ \forall(\sigma, \sigma') \in \alpha.disjoint(\{\sigma(x) \mid x \in \lambda\}) \wedge disjoint(\{\sigma'(x) \mid x \in \lambda'\}) \\ \forall(\sigma, \sigma') \in \alpha. \bigcup\{\sigma'(x) \mid x \in \lambda'\} \subseteq \bigcup\{\sigma(x) \mid x \in \lambda\} \end{array}}{\vdash A} \quad (\text{ACTION})$$

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

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

Figure 10: Type checking rules

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 get a collector that is on one hand easy to verify and on the 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 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) and it requires activating the write-

$$\frac{}{r; \{\} \vdash_p \{\phi\} skip \{\phi\}} \text{ (SKIP)} \qquad \frac{}{\mathrm{r}^-; \{\} \vdash_p \{\phi\} assert\ le \{\phi\}} \text{ (ASSERT1)}$$

$$\frac{}{\mathrm{r}^+; \{\} \vdash_p \{\phi \wedge le\} assert\ le \{\phi\}} \text{ (ASSERT2)}$$

$$\frac{}{r; \{\} \vdash_p \{e\} yield\ e, \lambda \{e\}} \text{ (YIELD)}$$

$$\frac{as(A) = (\rho, \alpha, m) \quad M \subseteq ThreadLocals \quad \alpha \Rightarrow Havoc(Globals \cup M \cup Locals) \quad \phi \Rightarrow \rho \quad Unsat(\phi \circ \alpha \circ \neg\psi)}{r; M \vdash_p \{\phi\} call\ A \{\psi\}} \text{ (ATOMIC)}$$

$$\frac{ps(P) = (\phi, M, \psi) \quad P \notin dom(rs)}{r; M \vdash_p \{\phi\} call\ P \{\psi\}} \text{ (PROC1)}$$

$$\frac{ps(P) = (\phi, M, \psi) \quad P \in dom(rs) \quad r; M \vdash_p \{\phi\} call\ rs(P) \{\psi\}}{r; M \vdash_p \{\phi\} call\ P \{\psi\}} \text{ (PROC2)}$$

$$\frac{ps(P) = (\phi, M, \psi)}{r; \{\} \vdash_p \{\rho \wedge \phi\} async\ P \{\rho\}} \text{ (ASYNC)}$$

$$\frac{r; M \vdash_p \{\phi_1 \wedge e\} s \{\phi_2\}}{r; M \vdash_p \{\phi_1 \wedge e\} ablock\ \{e, \lambda\}\ s\ \{\phi_2\}} \text{ (ABLOCK)}$$

$$\frac{r; M_1 \vdash_p \{\phi_1\} SS \{\phi_2\} \quad r; M_2 \vdash_p \{\phi_2\} s \{\phi_3\}}{r; M_1 \cup M_2 \vdash_p \{\phi_1\} SS; s \{\phi_3\}} \text{ (SEQ)}$$

$$\frac{r; M_1 \vdash_p \{e \wedge \phi_1\} s_1 \{\phi_2\} \quad r; M_2 \vdash_p \{\neg e \wedge \phi_1\} s_2 \{\phi_2\}}{r; M_1 \cup M_2 \vdash_p \{\phi_1\} if\ le\ then\ s_1\ else\ s_2\ \{\phi_2\}} \text{ (ITE)}$$

$$\frac{r; M \vdash_p \{e \wedge le\} s \{e\}}{r; M \vdash_p \{e\} while\ \{e, \alpha\}\ le\ do\ s\ \{e \wedge \neg le\}} \text{ (WHILE)}$$

$$\frac{r; M \vdash_p \phi \Rightarrow \phi' \quad r; M \vdash_p \{\phi'\} SS \{\psi'\} \quad \psi' \Rightarrow \psi}{r; M \vdash_p \{\phi\} SS \{\psi\}} \text{ (WEAKEN)}$$

$$\frac{r; M \vdash_p \{\phi\} SS \{\psi\} \quad Vars(\rho) \cap M = \{\}}{r; M \vdash_p \{\rho \wedge \phi\} SS \{\rho \wedge \psi\}} \text{ (FRAME)}$$

$$\frac{ps(P) = (\phi, M, \psi) \quad r = \mathrm{r}^+ \iff P \in dom(rs) \quad r; M' \vdash_p \{\phi\} bs(P) \{\psi\} \quad M' \subseteq M}{\vdash_p P} \text{ (PROCEDURE)}$$

$$\frac{r; M \vdash_p \{\phi'\} SS \{\psi\} \quad (Vars(\phi) \cup Vars(\psi)) \cap Locals = \{\} \quad \forall G, TL.\ \phi(G, TL) \Rightarrow \phi'(G, TL, L)}{r; M \vdash_p \{\phi\} (L, SS) \{\psi\}} \text{ (STACKFRAME)}$$

$$\frac{T = (TL, (L, SS)) \quad G \cdot TL \cdot L \vdash_p \phi \rightarrow true \quad \mathrm{r}^-; M \vdash_p \{\phi\} (L, SS) \{true\}}{G \vdash_p T} \text{ (THREAD)}$$

$$\frac{\forall P \in ProcName.\ \vdash_p P \quad \forall 1 \leq i \leq n.\ G \vdash_p T_i}{\vdash_p (bs, as, ps, rs, ls, G, T_1 \ldots T_n)} \text{ (PROGRAM)}$$

Figure 11: Sequential rules for partial correctness

$$\frac{}{P \vdash_r skip : \epsilon} \text{ (SKIP)} \qquad \frac{}{P \vdash_r assert\ le : \epsilon} \text{ (ASSERT)}$$

$$\frac{}{P \vdash_r yield\ e, \lambda : \epsilon} \text{ (YIELD)} \qquad \frac{P' \in dom(rs) \quad as(rs(P')) = (\rho', \alpha', m) \quad \rho' \circ \alpha' \Rightarrow Havoc(\{\})}{P \vdash_r call\ P' : \epsilon} \text{ (CALL-LOOP)}$$

$$\frac{P' \in dom(rs) \quad as(rs(P)) = (\rho, \alpha, m) \quad as(rs(P')) = (\rho', \alpha', m) \quad OldLocals = old(Locals) \quad \rho' \circ \alpha' \Rightarrow \exists OldLocals, Locals.\ \alpha}{P \vdash_r call\ P' : N} \text{ (CALL-ACTION)}$$

$$\frac{P' \in dom(rs) \quad as(rs(P')) = (\rho', \alpha', m) \quad \rho' \circ \alpha' \Rightarrow Havoc(\{\})}{P \vdash_r async\ P' : \epsilon} \text{ (ASYNC)}$$

$$\frac{s \preceq old(e) \Rightarrow Havoc(L)}{P \vdash_r ablock\ \{e, \lambda\}\ s : \epsilon} \text{ (ABLOCK-LOOP)}$$

$$\frac{as(rs(P)) = (\rho, \alpha, m) \quad OldLocals = old(Locals) \quad s \preceq old(e) \Rightarrow \exists OldLocals, Locals.\ \alpha}{P \vdash_r ablock\ \{e, \lambda\}\ s\ : N} \text{ (ABLOCK-ACTION)}$$

$$\frac{P \vdash_r s_1 : re_1 \quad P \vdash_r s_2 : re_2}{P \vdash_r s_1; s_2 : re_1 \cdot re_2} \text{ (SEQ)}$$

$$\frac{P \vdash_r s_1 : re_1 \quad P \vdash_r s_2 : re_2}{P \vdash_r if\ le\ then\ s_1\ else\ s_2\ : re_1 + re_2} \text{ (ITE)}$$

$$\frac{P \vdash_r s : re}{P \vdash_r while\ \{e, \alpha\}\ le\ do\ s\ : re^*} \text{ (WHILE)}$$

$$\frac{\forall P \in dom(rs).\ bs(P) \vdash_r \{N\}}{\vdash_r (bs, as, ps, rs, ls, G, T_1 \ldots T_n)} \text{ (PROGRAM)}$$

Figure 12: Refinement rules

$$\frac{}{skip \preceq Havoc(\{\})} \text{ (SKIP)} \qquad \frac{}{assert\ le \preceq Havoc(\{\})} \text{ (ASSERT)}$$

$$\frac{}{yield\ e, \lambda \preceq false} \text{ (YIELD)} \qquad \frac{as(A) = (\rho, \alpha, m)}{call\ A \preceq \alpha} \text{ (ATOMIC)}$$

$$\frac{}{call\ P \preceq false} \text{ (CALL)} \qquad \frac{}{async\ P \preceq Havoc(\{\})} \text{ (ASYNC)}$$

$$\frac{s \preceq \alpha}{ablock\ \{e, \lambda\}\ s\ \preceq \alpha} \text{ (ABLOCK)} \qquad \frac{s_1 \preceq \alpha_1 \quad s_2 \preceq \alpha_2}{s_1; s_2 \preceq \alpha_1 \circ \alpha_2} \text{ (SEQ)}$$

$$\frac{s_1 \preceq \alpha_1 \quad s_2 \preceq \alpha_2}{if\ le\ then\ s_1\ else\ s_2\ \preceq (le \circ \alpha_1) \vee (\neg le \circ \alpha_2)} \text{ (ITE)}$$

$$\frac{s \preceq \beta \quad \neg le \circ Havoc(\{\}) \Rightarrow \alpha \quad \beta \circ \alpha \Rightarrow \alpha}{while\ \{e, \alpha\}\ le\ do\ s\ \preceq e \circ \alpha \circ \neg le} \text{ (WHILE)}$$

$$\frac{s \preceq \alpha \quad \alpha \Rightarrow \alpha'}{s \preceq \alpha'} \text{ (WEAKEN)}$$

Figure 13: Abstracting statements by actions

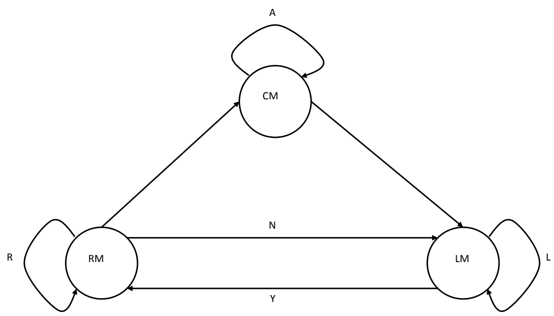

Figure 14: Specification for yield sufficiency

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

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

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

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

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

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

$$\frac{P \notin dom(rs)}{\vdash_y \mathit{call}\ P : (x, RM)}\ (\textsc{CallYield}) \qquad \frac{}{\vdash_y \mathit{async}\ P : (x, LM)}\ (\textsc{Async})$$

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

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

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

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

$$\frac{\vdash_y bs(P) : (x, y)}{\vdash_y P}\ (\textsc{Procedure}) \qquad \frac{\vdash_y SS : (x, y)}{\vdash_y (L, SS) : (x, y)}\ (\textsc{StackFrame})$$

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

$$\frac{\forall P \in ProcName \setminus dom(rs).\ \vdash_y P \qquad \forall 1 \le i \le n.\ \vdash_y T_i}{\vdash_y (bs, as, ps, rs, ls, G, T_1 \ldots T_n)}\ (\textsc{Program})$$

Figure 15: Yield sufficiency rules

$$\frac{}{rs; AS \vdash \mathit{skip} \leadsto \mathit{skip}}\ (\textsc{Skip}) \qquad \frac{}{rs; AS \vdash \mathit{assert}\ le \leadsto \mathit{assert}\ le}\ (\textsc{Assert})$$

$$\frac{}{rs; AS \vdash \mathit{yield}\ e, \lambda \leadsto \mathit{yield}\ e, \lambda}\ (\textsc{Yield})$$

$$\frac{A \notin AS}{rs; AS \vdash \mathit{call}\ A \leadsto \mathit{call}\ A}\ (\textsc{Atomic})$$

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

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

$$\frac{P \in dom(rs)}{rs; AS \vdash \mathit{call}\ P \leadsto \mathit{call}\ rs(P)}\ (\textsc{Proc1})$$

$$\frac{P \notin dom(rs)}{rs; AS \vdash \mathit{call}\ P \leadsto \mathit{call}\ P}\ (\textsc{Proc2})$$

$$\frac{}{rs; AS \vdash \mathit{async}\ P \leadsto \mathit{async}\ P}\ (\textsc{Async})$$

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

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

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

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

$$\frac{}{rs; AS \vdash (\rho, \alpha, m) \leadsto (\rho, \alpha, m)}\ (\textsc{Action})$$

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

$$\frac{\begin{array}{c} Prog = (bs, as, ps, rs, ls, G, T_1 \ldots T_n) \\ Prog' = (bs', as', ps', rs', ls', G, T_1' \ldots T_n') \\ \forall 1 \le i \le n.\ (rs; AS \vdash T_i \leadsto T_i') \\ \forall A \notin AS.\ (rs; AS \vdash as(A) \leadsto as'(A)) \\ \forall P \notin dom(rs).\ (rs; AS \vdash bs(P) \leadsto bs'(P)) \end{array}}{AS \vdash Prog \leadsto Prog'}\ (\textsc{Program})$$

Figure 16: Program refinement

barrier on writes to root pointers, which means write barrier overhead on the runtime stack accesses and register accesses. The first issue means that the collector is unacceptable for use with modern multicore platforms. The second issue implies bad performance. Concurrent garbage collectors today are expected to work correctly without adding a write-barrier overhead to local accesses.

We therefore extended and modified Dijkstra's collector to make it work with parallel programs and also not require applying a write-barrier on root modifications. As far as we know, this garbage collector's algorithm has not been proposes previously in the literature.

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.