Collecting garbage concurrently (but correctly)

Kamal Lodaya
The Institute of Mathematical Sciences, Chennai

Joint work
with ***Kalpesh Kapoor (IIT, Guwahati)***
and ***Uday Reddy (U. Birmingham)***

First order logic (Dedekind 1888, Frege 1893)

Terms are built up from variables and constant symbols using function symbols (eg, numbers with $+, \times$)

Atomic formulas have an n-ary predicate symbol applied to n terms eg, the binary relational operators $<, \leq, >, \geq, =, \neq$

The constant, function and predicate symbols are called a signature. Equality is always assumed to be in the signature

Formulas are built up from atomic formulas using the boolean operations $\wedge, \vee, \neg$ and quantifiers $\forall x, \exists x$

A model/structure/interpretation $(D, \mathcal{I})$ is a nonempty set (domain), and an interpretation of the symbols in the signature
eg, $\mathcal{N} = (\mathcal{N}, plus, times)$, the natural numbers with the usual addition and multiplication

Truth and validity

Let s be a state, a function assigning values in the domain to variables

This inductively extends to $\widehat{s}$ assigning values to terms
eg, $\widehat{s}(t_1 \times t_2) = \widehat{s}(t_1)$ times $\widehat{s}(t_2)$

This further extends to a satisfaction/truth definition $(D, \mathcal{I}), s \models p$ (sometimes written $(D, \mathcal{I}) \models p[s]$)

eg, $(D, \mathcal{I}), s \models p \wedge q$ iff $(D, \mathcal{I}), s \models p$ and $(D, \mathcal{I}), s \models q$

A formula is valid if it is true in all models and in all states. (Sometimes validity is only defined for sentences—formulas without free variables.)

Theorem 1 (Turing 1936) *Validity of FO formulas is undecidable.*

Validity over interpretations

We can restrict to particular interpretations:
eg, $FO(\mathcal{N})$ and $FO(\mathcal{Z})$.
The sets of valid formulas of these interpretations are written $Th(\mathcal{N})$ and $Th(\mathcal{Z})$.

If a formula p is valid in an interpretation $\mathcal{I}$, we call it a consequence of $Th(\mathcal{I})$ and write $Th(\mathcal{I}) \models p$.

Theorem 2 (Turing 1936) *$Th(\mathcal{N})$ and $Th(\mathcal{Z})$ are not even computably enumerable.*

We can also restrict to particular signatures:

Theorem 3 (Presburger 1929) *The set $Th(\mathcal{N}, plus)$ is decidable.*

Truth and proof

An axiom system gives syntactic ways of deriving validities or consequences.

Let Γ be a set of premisses (eg, it could be $Th(\mathcal{N})$).

Here is an inference rule: $\dfrac{\Gamma \vdash p, \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q}$ and-intro

An axiom system is sound if it can only derive validities/ consequences; complete if it can derive all validities/consequences.

Theorem 4 (Completeness, Gödel 1930) *There is a finite axiomatization of FO. Hence its valid formulas are computably enumerable/r.e.*

Theorem 5 (Incompleteness, Gödel 1931) *There is no finite axiomatization for $FO(\mathcal{N})$ and $FO(\mathcal{Z})$.*

Program assertions (Naur 1966, Floyd 1967)

Program variables hold values which are integers/reals/etc

We will assume a fixed interpretation $FO(\mathcal{N}, plus, times)$.
The satisfaction relation is just $s \models p$.
A formula is valid if it is true in all states.

We could define an "additive" proof theory like $\dfrac{s \vdash p, \quad s \vdash q}{s \vdash p \wedge q}$

Logic of programs (Hoare 1969)

A logic of programs is a multi-modal logic with countably infinite modalities $\{\Box_{x,e}p \mid x \in Var, e \in Exp\}$.

A logic of programs is a restricted multi-modal logic with formulae $\{p \supset q,\ p \supset [\mathtt{x\ :=\ e}]q \mid x \in Var, e \in Exp,\ p, q \in FO\}$.

A logic of programs has formulae of two kinds: implications $p \supset q$ and "correctness triples" $\{p\}\mathtt{x:=e}\{q\}$.

The usual modal consequence

$p \supset p',\ q' \supset q, p' \supset \Box_{x,e}q' \vdash p \supset \Box_{x,e}q)$ K+Necessitation

is written

$$\frac{Th(\mathcal{I}) \vdash p \supset p', \quad \{p'\}\mathtt{x:=e}\{q'\}, \quad Th(\mathcal{I}) \vdash q' \supset q}{\{p\}\mathtt{x:=e}\{q\}}$$ consequence

There is also one axiom:

$\{q[e/x]\}\mathtt{x\ :=\ e}\{q\}$ assignment

LOGIC OF PROGRAMS (Hoare 1969)

To verify a structured program, we associate a correctness triple $\{p\}$ `C` $\{q\}$ with every construct `C` in the program.

eg, $\{x > 0\}$ `x := x-1` $\{x \geq 0\}$

The triple is valid ($\models \{p\}C\{q\}$) if
for all states $s \models p$, when $s[\![C]\!]t$ then $t \models q$

STRUCTURED PROGRAMS

$$\{q[e/x]\}\texttt{x := e}\{q\}$$ assignment

$$\frac{\{p\}C_1\{q\},\quad \{q\}C_2\{r\}}{\{p\}C_1;C_2\{r\}}$$ sequencing

$$\frac{\{p \wedge b\}C_1\{q\},\quad \{p \wedge \neg b\}C_2\{q\}}{\{p\}\texttt{if } b \texttt{ then } C_1 \texttt{ else } C_2 \texttt{ end if } \{q\}}$$ if

$$\frac{\{q \wedge b\}C\{q\}}{\{q\}\texttt{while } b \texttt{ do } C\{q \wedge \neg b\}}$$ while

Details

The proof theory is relative to an interpretation:

$$\frac{Th(\mathcal{I}) \vdash p \supset p', \quad \{p'\}C\{q'\}, \quad Th(\mathcal{I}) \vdash q' \supset q}{\{p\}C\{q\}} \qquad \text{consequence}$$

Some structural rules are also needed:

$$\frac{\{p_1\}C\{q_1\}, \quad \{p_2\}C\{q_2\}}{\{p_1 \wedge p_2\}C\{q_1 \wedge q_2\}} \qquad \text{conjunction}$$

Dijkstra 1975 introduced the idea of weakest preconditions

Theorem 6 (Cook 1978) *If $Th(\mathcal{I})$ can express weakest preconditions and loop invariants, then Hoare logic is relatively complete wrt $Th(\mathcal{I})$.*

Also works for disjoint concurrency. Assume the proviso that for processes $i \neq j$, $write(C_i) \cap free(p_j, C_j, q_j) = \emptyset$.

$$\frac{\{p_1\}C_1\{q_1\}, \quad \{p_2\}C_2\{q_2\}}{\{p_1 \wedge p_2\}C_1 || C_2\{q_1 \wedge q_2\}} \qquad \text{disjoint parallel}$$

Verification of programs with data structures

Data structures require treating the state as divided into many parts

Burstall 1969 pointed out that you need inductive predicates over the parts to state properties of the whole (eg, lists)

Manna and Waldinger 1985, 1990 present theories for inducting over commonly used data types, and techniques which theorem provers can employ to use them. For example, in proving programs with lists, we might use the inductive predicate:

$list\ \varepsilon\ i \stackrel{\mathrm{def}}{=} i = nil$, and
$list\ aV\ i \stackrel{\mathrm{def}}{=} head(i) = a \wedge list\ V\ tail(i)$

Programs with pointers

A heap is a function from addresses to values. The addresses can be computed inside a program. We let the set of addresses be $\mathcal{N}$. Given two disjoint heaps h_1 and h_2, $h_1 \circ h_2$ is their fusion.

$x := \texttt{cons}(e_1, \ldots, e_n)$ allocates n consecutive locations from the heap and stores the values of the expressions in them

The expression $[e]$ computes the value of e as a heap address, and returns the record/field at that address

$[e] := e'$ computes the value of e as a heap address, and updates it with the value of e'

$\texttt{dispose}\ e$ computes the value of e as a heap address, and returns that address to the heap

If the value of e in the last three commands is not a heap address, the program aborts. A program is safe if it does not abort.

Logic of programs with pointers (Reynolds 2000)

Atomic formulas include a binary predicate symbol $\longmapsto$ and a constant symbol emp

Formulas are built up from atomic formulas using the boolean operations $\wedge, \vee, \neg$ and quantifiers $\forall x, \exists x$,
and a binary operation $\star$

A state (s, h) consists of a stack and a heap

$s, h \models emp$ iff $dom(h) = \emptyset$

$s, h \models t_1 \longmapsto t_2$ iff $dom(h) = \{\widehat{s}(t_1)\}$ and $h(\widehat{s}(t_1)) = \widehat{s}(t_2)$

$s, h \models p \star q$ iff $\exists h_1, h_2 (h = h_1 \circ h_2$ and $s, h_1 \models p$ and $s, h_2 \models q)$

A formula is valid if it is true in all states and heaps

The proof theory of heaps is "multiplicative": $\dfrac{h_1 \vdash p, \quad h_2 \vdash q}{h_1 \circ h_2 \vdash p \star q}$

Some formulae

Precise assertions

p is precise iff for all s, h, there is at most one $h_0 \subseteq h : s, h_0 \models p$

$x \longmapsto e$

$x \longmapsto (e_0, e_1, \ldots, e_n) \stackrel{\text{def}}{=} (x \longmapsto e_0) \star (x + 1 \longmapsto e_1) \star \ldots \star (x + n \longmapsto e_n)$

$x \longmapsto \_ \stackrel{\text{def}}{=} \exists y (x \longmapsto y)$ (don't care)

Imprecise assertions

$\exists x (x \longmapsto 10)$ (don't know)

$x \hookrightarrow e \stackrel{\text{def}}{=} (x \longmapsto e) \star \mathit{true}$ ($\mathit{dom}(h)$ can have other elements)

$(p_1 \wedge p_2) \star q \supset (p_1 \star q) \wedge (p_2 \star q)$ is a valid formula.

$(p_1 \star q) \wedge (p_2 \star q) \supset (p_1 \wedge p_2) \star q$ is valid when q is precise.

Exercise: Compare $x \longmapsto (3, y) \star y \longmapsto (3, x)$ with $x \longmapsto (3, y) \wedge y \longmapsto (3, x)$ and $x \hookrightarrow (3, y) \wedge y \hookrightarrow (3, x)$.

Axiom system (Reynolds 2000, Ishtiaq and O'Hearn 2001)

There are three axioms for the atomic statements.

$\{emp\}x := \texttt{cons}(e_1, \ldots, e_n)\{x \longmapsto (e_1, \ldots, e_n)\}$ allocation

$\{e_1 \longmapsto \_\}[e_1] := e_2\{e_1 \longmapsto e_2\}$ mutation

$\{e \longmapsto \_\}\texttt{dispose}\ e\{emp\}$ deallocation

Ishtiaq and O'Hearn 2001 introduced a structural rule:

$$\frac{\{p\}C\{q\}, \quad write(C) \cap free(r) = \emptyset}{\{p \star r\}C\{q \star r\}}$$ frame rule

$\models \{p\}C\{q\}$ if for all states and heaps $s, h \models p$,
C is safe at (s, h) and when $(s, h)[\![C]\!](s', h')$, then $s', h' \models q$

With the proviso that for processes $i \neq j$,
$write(C_i) \cap free(p_j, C_j, q_j) = \emptyset$:

$$\frac{\{p_1\}C_1\{q_1\}, \quad \{p_2\}C_2\{q_2\}}{\{p_1 \star p_2\}C_1||C_2\{q_1 \star q_2\}}$$ disjoint parallel

List reversal: top level proof

$\star$ has separation built-in, and we can define precise assertions:
$list\ \varepsilon\ i \stackrel{\text{def}}{=} i = nil$ and $list\ aV\ i \stackrel{\text{def}}{=} \exists j (i \longmapsto (a, j) \star list\ V\ j)$.

pre $\{list\ V\ i\}$

```
j := nil;
```

inv $\{\exists W, X((list\ W\ i \star list\ X\ j) \wedge V^R = W^R X)\}$

```
while i <> nil do
```

$\{\exists a, W, X((list\ aW\ i \star list\ X\ j) \wedge V^R = (aW)^R X)\}$

```
k := [i.next];
[i.next] := j;
j := i;
i := k
```

inv $\{\exists W, X((list\ W\ i \star list\ X\ j) \wedge V^R = W^R X)\}$

```
end while
```

post $\{list\ V^R\ j\}$

List reversal: proof of loop body

pre $\{\exists a, W, X((list\ aW\ \ i \star list\ \ X\ \ j) \wedge V^R = (aW)^R X)\}$

$\{\exists a, W, X, k((i \longmapsto a, k \star list\ W\ \ i \star list\ \ X\ \ j) \wedge V^R = (aW)^R X)\}$

```
k := [i.next];
```

$\{\exists a, W, X((i \longmapsto a, k \star list\ W\ \ k \star list\ \ X\ \ j) \wedge V^R = (aW)^R X)\}$

```
[i.next] := j;
```

$\{\exists a, W, X((i \longmapsto a, j \star list\ W\ \ k \star list\ \ X\ \ j) \wedge V^R = (aW)^R X)\}$

$\{\exists a, W, X((list\ W\ \ k \star list\ aX\ \ i) \wedge V^R = W^R aX)\}$

$\{\exists W, X((list\ W\ \ k \star list\ \ X\ \ i) \wedge V^R = W^R X)\}$

```
j := i; i := k
```

inv $\{\exists W, X((list\ W\ \ i \star list\ \ X\ \ j) \wedge V^R = W^R X)\}$

DARING CONCURRENCY (O'Hearn 04)

```
x :=                  [y] :=
cons(...);      ||    ...;
[x] := ...            dispose (y)
```

O'Hearn calls this daring —as opposed to the cautious concurrency enforced by a programming discipline like a monitor.

There is potential aliasing between x and y, but when proving the parallel program, we use the $\star$ operation to rule out race conditions.

The process which allocates owns the storage allocated by a pointer. This ownership can be transferred by passing the pointer, but guaranteeing single ownership allows use of the disjoint concurrency rule. Ownership is determined by the assertions, not by the program.

But what if we want to prove programs with race conditions?

Collecting garbage: Mutator

$$\text{mutator} \stackrel{\text{def}}{=}$$

```
var k,j,f: unsigned;
do
    true ⇒ delete left edge(k);
□  true ⇒ delete right edge(k);
□  true ⇒ modify left edge(k,j);
□  true ⇒ modify right edge(k,j);
□  true ⇒ get new left edge(k);
□  true ⇒ get new right edge(k);
od
```

Collector (mark and sweep)

```
collector ≝
        var i: unsigned; c: (white,black);
        do true ⇒
                mark;
                sweep
        od
```

Marking

mark $\stackrel{\text{def}}{=}$

blacken(ROOT); blacken(NIL);

i := 0;

```
do i ≤ N ⇒
        if [i.colour] = white ⇒ i := i+1
        □ [i.colour] = black and
          [[i.left].colour] = black and
          [[i.right].colour] = black ⇒ i := i+1
        □ else ⇒ blacken(i.left); blacken(i.right);
                 blacken(i); i:= 0
        fi
od
```

Sweeping

$$
\begin{array}{l}
\text{sweep} \;\stackrel{\text{def}}{=} \\
\qquad \text{i} := 0; \\
\qquad \texttt{do}\ \text{i} \leq \text{N} \;\Rightarrow \\
\qquad\qquad \texttt{if}\ [\text{i.colour}] = \text{black} \;\Rightarrow \text{whiten(i); i:= i+1} \\
\qquad\qquad \square\ [\text{i.colour}] = \text{white} \;\Rightarrow \text{collect white node(i);} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{i := i+1} \\
\qquad\qquad \texttt{fi} \\
\qquad \texttt{od}
\end{array}
$$

Collector proof outline

collector $\stackrel{\text{def}}{=}$

```
var i: unsigned; c: (white,black);
do true  ⇒ {no black nodes}
            mark;
            {all white nodes are garbage}
            sweep
            {no black nodes}
od
```

Concurrent garbage collector (Steele)

Two bits with every memory location, for "marked" and "swept"

Multiprocessing compactifying garbage collection,
by Guy L. Steele Jr.,
Comm.ACM (submitted Sep 74, published Sep 75)
ACM Student Award, 1st place

Implementation

```
mark ≝
    blacken(ROOT); blacken(NIL);
    i := 0;
    do i ≤ N ⇒
              if [i.colour] = white ⇒ i := i+1
              □ [i.colour] = black and
                [[i.left].colour] = black and
                [[i.right].colour] = black ⇒ i := i+1
              □ else ⇒ blacken(i.left); blacken(i.right);
                        blacken(i); i:= 0
              fi
    od
```

Implementation questions

- How is the test supposed to be done atomically?
- What if mutator is modifying while collector is blackening?

Dijkstra, EWD 492, Apr 75 introduced the idea of an intermediate gray colour. He says:

A.J.Martin and E.F.M.Steffens selected and formulated the ... problem and did most of the preliminary investigations.

I arrived at its solution during a discussion with the latter, W.H.Feijen and M.Rem.

It is a pleasure to acknowledge their share in its discovery.

Bug (Stenning 75)

addleft(p,q):

$\{\exists U : reachGraph(U) \wedge p \hookrightarrow (l, m, \_) \wedge p \neq nil \wedge q \in U\}$

$\langle$atleastgrey(q); $\rangle$

$\langle$[p.left] := q $\rangle$

$\{\exists U : reachGraph(U) \wedge p \hookrightarrow (q, m, \_) \wedge q \in U\}$

$\langle C \rangle$ says the commands in C have to be done as an atomic action.

$reachGraph(U)$ says U is the set of nodes reachable from *root*.

$p \hookrightarrow (l, m, c)$ is $[p.left] = l \wedge [p.right] = m \wedge [p.colour] = c$.

Fix

The bug is fixed by greying the node *after* the mutation.

addleft(p,q):

$\{\exists U : reachGraph(U) \wedge p \hookrightarrow (l, m, \_) \wedge p \neq nil \wedge q \in U \wedge mod = nil\}$

$\langle$[p.left] := q; *mod*:= p; $\rangle$

$\{\exists U : reachGraph(U) \wedge p \hookrightarrow (q, m, \_) \wedge q \in U \wedge mod = p \neq nil\}$

$\langle$atleastgrey(q); *mod*:= NIL $\rangle$

$\{\exists U : reachGraph(U) \wedge p \hookrightarrow (q, m, \_) \wedge q \in U \wedge mod = nil\}$

Different kinds of proof

DLMSS proof: global invariant proof, insightful but informal and known to be unreliable.

Gries proof: non-interference proof, which is formal, hence mechanizable, but not compositional.

Our proof: global invariant proof, formal, compositional, hence easier to comprehend.

Another approach

Vafeiadis and Parkinson 07 use a rely-guarantee proof system which is also formal and compositional. We have not proved this program using their proof system.

Global invariants for shared store (O'Hearn 04)

The basic idea is to treat every atomic command as "grabbing" the shared variables that it requires, assuming that a resource invariant holds in the beginning and re-establishing at the end. This is described by the rule:

$$\frac{\{p \star RI\}C\{q \star RI\}}{RI \vdash \{p\} < C > \{q\}} \qquad \text{atomic share}$$

where C is an atomic command and the free variables of p or q are not modified in other processes. (But the free variables of RI can be.)

$$\frac{RI \vdash \{p_1\}C_1\{q_1\}, \quad \ldots, \quad RI \vdash \{p_n\}C_n\{q_n\}}{RI \vdash \{p_1 \star \ldots \star p_n\}C_1 || \ldots || C_n\{q_1 \star \ldots \star q_n\}}, \qquad \text{shared parallel}$$

where no local variable free in p_i or q_i is changed in C_j, for $i \neq j$ in $\{1, \ldots, n\}$. (But the free variables of RI can be.)

Permission algebra (Bornat, Calcagno, O'Hearn, Parkinson 05)

Full permission (for reading as well as writing on a heap location) is denoted 1, read access permission is denoted R, and the complement of a read permission is denoted $-R$. A read permission and its complement can be combined to obtain a full permission $R \star (-R) = 1$. Both R and $-R$ permissions allow reading, but only 1 allows writing (in addition to reading).

The axioms for reading and writing heap locations are:

$$\{e \xmapsto{p} e'\} x := [e] \{e \xmapsto{p} e' \wedge x = e'\} \qquad \text{read}$$

$$\{e \xmapsto{1} \_\} [e] := e' \{e \xmapsto{1} e'\} \qquad \text{mutation}$$

where p is either R or $-R$.

Using the frame rule of separation logic, we can also conclude

$$\{e \xmapsto{1} e'\} x := [e] \{e \xmapsto{1} e' \wedge x = e'\}.$$

Permissions with a resource invariant

In addition to the processes themselves, permissions are also deposited in the resource invariant.

When accessing a resource (shared storage), a process grabs the heap cells described by the invariant along with their permissions: the conjunction $(i \xmapsto{p} j) \star (i \xmapsto{q} j)$ is equivalent to providing access $i \xmapsto{p \star q} j$.

Explaining the resource invariant would take several slides, but different parts of it can be modularly understood. The proof outlines involve detailed combinatorial reasoning.

The original program with the bug *cannot* be proved correct.