Extending ITL with Interleaved Programs for Interactive Verification

Gerhard Schellhorn

joint work with

Bogdan Tofan, Gidon Ernst, Kurt Stenzel,
Wolfgang Reif, Michael Balser, Simon Bäumler

Institute for Software and Systems Engineering
University of Augsburg

TIME, Lübeck, 13.9.2011

Background: Development of Correct Software

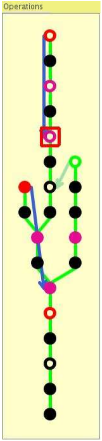
General Setting:

- Specification of Software Systems with:
  Algebraic Specification, Z,
  Abstract State Machines (ASMs)
- Incremental Refinement of Designs:
  Algebraic, Data, ASM Refinement
- Verification of refinements:
  Tool support with KIV Interactive Verifier

Background: Proving Sequential Programs with KIV

KIV is an interactive theorem prover based on

- Structured algebraic specification of data types with higher-order logic
- Sequent calculus with proof trees
- wp-calculus for ASMs and Java
- Proof principle for sequential programs: symbolic execution (+ induction) [Burstall 74] (= incremental computation of strongest postconditions for instructions)



Concurrent systems: What Logic to use?

Define a general logic which

- allows proofs for arbitrary properties: safety, liveness, deadlock, fairness, refinement (trace inclusion)
- can handle systems specifications
  that use abstract data types
  $\Rightarrow$ interactive proving approach
- provides modular support for various forms of concurrency:
  Programs with interleaving (“threading”)
  Synchronous and asynchronous programs
  Harel- and UML-Statecharts
  (no encoding to transition systems)

Concurrent systems: What Calculus to use?

Define a calculus where

- proving properties (e.g. contracts) for sequential programs should not be more difficult than using wp-calculus
- compositional reasoning (e.g. rely-guarantee) is supported, as otherwise concurrency generates too many cases

Content of my talk:

- One particular answer to choosing a logic and a calculus, based on ITL [Moszkowski 00].
- Some applications for interleaved programs.

Outline

- The Logic RGITL
  - Compositional interleaving
  - A semantics with system and environment steps
  - Integration with HOL
- Proof principles in RGITL
  - Symbolic Execution
  - Induction
  - Rely-Guarantee
- Application: Lock-Free Algorithms
  - Motivation
  - Simple Example: Treiber's Stack
  - Linearizability and Lock-Freedom
- Experiences, Future Work

Why base the logic on ITL?

+ ITL directly offers termination/nontermination by using finite & infinite intervals
+ ITL is (easily) compatible with higher-order logic.
+ ITL offers the concept: programs $\subseteq$ formulas. The semantics of both is a set of intervals.
– Some small extensions are needed:
  Is variable M in the program N := t?
  Recursive procedures
– ITL does not offer a concept for interleaving.

Interleaving: Informal Semantics

Interleaved program $\{N := N^2; N := N^2\} \parallel N := N + 1$ started with $N = 2$:

$$
\begin{tikzcd}
N = 17 & & N = 25 & & N = 81 \\
N = 16, N := N + 1 \arrow[u, "STEP"] & & N := N^2, N = 5 \arrow[u, "STEP"] & & N = 9, N := N^2 \arrow[u, "STEP"] \\
& N = 4, N := N^2 \parallel N := N + 1 \arrow[ul, "STEP"] \arrow[ur, "STEP"'] & & & N = 3, N := N^2; N := N^2 \arrow[u, "STEP"] \\
& & N = 2, \{N := N^2; N := N^2\} \parallel N := N + 1 \arrow[ul, "STEP"] \arrow[urr, "STEP"'] & &
\end{tikzcd}
$$

Weak Fairness:

$\{$**while** $N \neq 0$ **do** $N := N + 1\} \parallel N := 0$ terminates

Interleaving and Compositionality

A substitution rule is basic for a calculus to scale:

$$\frac{\alpha \to A \qquad \beta \to B \qquad A \oplus B \to C}{\alpha \oplus \beta \to C}$$

- holds in ITL for $\oplus$ = sequential composition and other operators (similar to Hoare calculus)
- ideally, third premise should be trivial
- should hold for $\oplus$ interleaving too!

Example for Noncompositional Interleaving in ITL

In classical ITL:

$$\{\textbf{while}^* \ N \neq 0 \ \textbf{do} \ N := 0\} \leftrightarrow \{\textbf{if}^* \ N \neq 0 \ \textbf{then} \ N := 0\} \qquad (1)$$

(the star indicates, that the test does not take time)

Example for Noncompositional Interleaving in ITL

In classical ITL:

$$\{\textbf{while}^* \ N \neq 0 \ \textbf{do} \ N := 0\} \leftrightarrow \{\textbf{if}^* \ N \neq 0 \ \textbf{then} \ N := 0\} \qquad (1)$$

(the star indicates, that the test does not take time)

Using the substitution rule:

$$\{\textbf{while}^* \ N \neq 0 \ \textbf{do} \ N := 0\} \parallel \{\textbf{while}^* \ N \neq 1 \ \textbf{do} \ N := 1\} \qquad (2)$$

$$\leftrightarrow$$

$$\{\textbf{if}^* \ N \neq 0 \ \textbf{then} \ N := 0\} \parallel \{\textbf{while}^* \ N \neq 1 \ \textbf{do} \ N := 1\} \qquad (3)$$

which is wrong:
(2) has nonterminating runs, which alternate between the loops
(3) terminates, since at some time N := 0 is executed

Example for Noncompositional Interleaving in ITL

In classical ITL:

$\{\textbf{while}^* \ N \neq 0 \ \textbf{do} \ N := 0\} \leftrightarrow \{\textbf{if}^* \ N \neq 0 \ \textbf{then} \ N := 0\}$ (1)

(the star indicates, that the test does not take time)

Using the substitution rule:

$\{\textbf{while}^* \ N \neq 0 \ \textbf{do} \ N := 0\} \parallel \{\textbf{while}^* \ N \neq 1 \ \textbf{do} \ N := 1\}$ (2)

$\leftrightarrow$

$\{\textbf{if}^* \ N \neq 0 \ \textbf{then} \ N := 0\} \parallel \{\textbf{while}^* \ N \neq 1 \ \textbf{do} \ N := 1\}$ (3)

which is wrong:

(2) has nonterminating runs, which alternate between the loops

(3) terminates, since at some time N := 0 is executed

The problem is, that equivalence (1) ignores effects of the environment of the program

Universität Augsburg

RGITL: Intervals with Environment Steps

Basic idea: environment steps between program steps

- Semantics is based on Intervals I = sequence of states (I(0), I′(0), I(1), I′(1), . . .)
- state = valuation of variables
- I has finite (termination!) or infinite length $\# I \in \mathbb{N} \cup \{\infty\}$
- I alternates system steps (I(0),I′(0)), (I(1),I′(1)), . . .
  with environment steps (I′(0),I(1)), (I′(1),I(2)), . . .
  (similar to reactive sequences [deRoever 01])
- Programs determine system steps only
- Primed and double primed (flexible) variables are needed:
  X, X′, X″ denote the value of X in I(0), I′(0), I(1)
  (X = X′ = X″ in final states by convention)

Semantics of the Example in RGITL

The semantics of **while*** $N \neq 0$ **do** $N := 0$ now are intervals where N has values ($n_i \neq 0$):

- (0)
- $(n_0, 0, 0)$ /* first env step does not change N */
- $(n_0, 0, n_1, 0, 0)$ /* env sets N to n_1 */
- $(n_0, 0, n_1, 0, n_2, 0, 0)$
- $\ldots$
- Nonterminating run $(n_0, 0, n_1, 0, n_2, 0, \ldots)$
- $\Rightarrow$ The two programs are not equivalent
- But: equivalence is provable with environment assumption:
  $(\Box\ N'' = N') \rightarrow$
  $(\{\textbf{while}^*\ N \neq 0\ \textbf{do}\ N := 0\} \leftrightarrow \{\textbf{if}^*\ N \neq 0\ \textbf{then}\ N := 0\})$

RGITL: Syntax

Extends simply types lambda-expressions with

- static (x) and flexible variables (X,X′,X″)
- formulas (= expressions of type bool) with:
  $\Diamond$, $\Box$, **until**, **A**, **E** /* all paths/exists path */,
  $\circ$, $\bullet$ /* strong/weak next state */,
  last /* termination */, **;** /* chop */, ***** /* star */
  $\|$, $\|_{nf}$ /* weak fair/nonfair interleaving */,
  $p(\underline{T};\underline{Y})$ /* procedure call with input an in-out parameters */

TL and HOL operators can be freely mixed

RGITL: Semantics

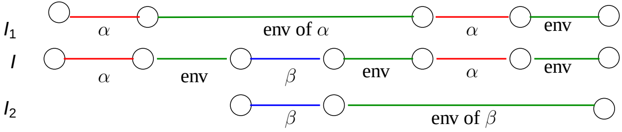
- Expressions are evaluated over algebras (constructed as models of algebraic specs.) and an interval I = (I(0),I′(0),I(1),. . .)
- If formula φ evaluates to true, write: I $\models \varphi$
- TL Operators have standard semantics:
  - (I(0), I′(0), I(1), I′(1), . . .) $\models \Box\, \varphi$ iff for all $n \leq$ # I: (I(n), I′(n), I(n + 1), I′(n + 1), . . .) $\models \varphi$
  - I $\models$ **A** φ iff for all J with J(0) = I(0): J $\models \varphi$
  - I $\models$ **last** iff I = (I(0))
  - (I(0), I′(0), . . .) $\models \exists$ X. φ iff ex. $(a_0, a_0', \ldots)$ with (I(0)[X $\leftarrow a_0$], I′(0)[X $\leftarrow a_0'$], . . .) $\models \varphi$

Programs in RGITL

- Programs α are formulas too:
  I $\models \alpha \Leftrightarrow$ the system steps in I are possible steps of α
- Programs: parallel assignments $\underline{X} := \underline{T}$,
  sequential (**let**, **while**, **or**, **choose**, rec. procedures) +
  $\alpha \parallel \beta$ (interleaving), **await** C (block until C holds)
- Programs α are placed in a frame assumption $[\alpha]_{X,Y}$
  to indicate which variables are fixed in assignments
  (similar to TLA [Lamport 94], but no built-in stuttering)
- $[X := T]_{X,Y} \leftrightarrow X' = T \land Y' = Y \land \circ$ **last**
- Typical goal: $\alpha \land E \to P$
  "Executing α in environment E satisfies P"

Semantics of Interleaving

- Interleaving of two programs (or formulas) α and β is defined compositionally, by interleaving individual intervals $\Rightarrow$ substitution rule is valid!
- Assume $\mathsf{I}_1 \models \alpha$, $\mathsf{I}_2 \models \beta$
- Interleaving gives all intervals I which have
  - Interleaved system steps from I_1 and I_2 (fair)
  - The environment steps of I_1 (I_2) are the relevant alternating sequences of env. steps and system steps of β (α) in I



- Formal def. in paper, including **blocked** steps (tricky):
  await $\varphi \equiv$ **while*** $\neg\, \varphi$ **do blocked**

Outline

- The Logic RGITL
  - Compositional interleaving
  - A semantics with system and environment steps
  - Integration with HOL
- Proof principles of RGITL
  - Symbolic Execution
  - Induction
  - Rely-Guarantee
- Application: Lock-Free Algorithms
  - Motivation
  - Simple Example: Treiber's Stack
  - Linearizability and Lock-Freedom
- Experiences, Future Work

Proof principle 1: Symbolic Execution

- Symbolic execution = Step forwards through an interval
- Advantage: no encoding of programs as transition systems with program counters (as in Step, TLA or Model checking) $\Rightarrow$ readable goals
- Symbolic execution is done in two phases: Unwinding and Stepping to the next state

Symbolic Execution: Unwinding (1)

- Splits formulas φ with $\underline{X}$ = free(φ) into formulas
  - p($\underline{X}$,$\underline{X}'$,$\underline{X}''$) describing the first step
  - $\circ\,\psi$ describing properties of the rest of the run
- Termination gives formulas of the form q($\underline{X}$) $\wedge$ **last**
- examples:

$$\Box\,\varphi \equiv \varphi \wedge \bullet\,\Box\,\varphi$$

$$\bullet\,\varphi \equiv \textbf{last} \vee \circ\,\varphi$$

$$[X := T; \alpha]_{X,\underline{Y}} \equiv X' = T \wedge \underline{Y}' = \underline{Y} \wedge \circ\,[\alpha]_{X,\underline{Y}}$$

$$[\textbf{let}\ X = T\ \textbf{in}\ \alpha]_{\underline{Y}} \equiv \exists\, X.(X = T \wedge [\alpha]_{X,\underline{Y}} \wedge \Box\, X' = X'')$$

$$\begin{array}{rcl} [\textbf{choose}\ X\ \textbf{with}\ \psi & \equiv & \exists\, X.(\psi \wedge [\alpha]_{X,\underline{Y}} \wedge \Box\, X' = X'') \\ \textbf{in}\ \alpha\ \textbf{ifnone}\ \beta]_{\underline{Y}} & & \vee\, (\neg\,\exists\, X.\psi) \wedge [\beta]_{\underline{Y}} \end{array}$$

Universität Augsburg

Symbolic Execution: Unwinding (2)

To unwind interleaving and compounds unwind subprograms:

- If $\alpha \equiv p(X, X', X'') \wedge \circ\, \alpha'$ then

$$\begin{aligned} \{\alpha; \beta\} &\equiv p(X, X', X'') \wedge \circ\, \{\alpha'; \beta\} \\ \{\alpha \parallel \beta\} &\equiv \{\alpha \mathrel{<\!\parallel} \beta\} \vee \{\alpha \mathrel{\parallel\!>} \beta\} \\ \{\alpha \mathrel{<\!\parallel} \beta\} &\equiv p(X, X', X'') \wedge \circ\, \{\alpha' \mathrel{<\!\parallel} \beta\} \end{aligned}$$

- If $\alpha \equiv q(X) \wedge \textbf{last}$ then

$$\begin{aligned} \{\alpha; \beta\} &\equiv q(X) \wedge \beta \\ \alpha \mathrel{<\!\parallel} \beta &\equiv q(X) \wedge \beta \end{aligned}$$

Symbolic Execution: Stepping

- Stepping removes the first step of interval:
  Instead of (I(0),I′(0),I(1),I′(1),...) consider (I(1), I′(1),...)
- Use new static variables x_0, x_1 to store I(0)(X) and I′(0)(X) of the old first step in I(1)(x_0) and I(1)(x_1)

$$\frac{p(x_0, x_1, X) \land \psi}{p(X, X', X'') \land \circ\, \psi}\ \textit{step} \qquad\qquad \frac{q(x_0)}{q(X) \land \textbf{last}}\ \textit{last}$$

- Effect: computation of the strongest postcondition of the first statement, weakened with environment assumption
  $\Rightarrow$ sequential programs are executed as in wp-calculus
- Temporal properties result in (often non-temporal) additional goals for intermediate states

Proof principle 2: Induction

- Proofs use induction over well-founded orders
- Temporal induction reduced to well-founded induction by:
  $\Diamond\, \varphi \equiv \exists\, N.\, N = N'' + 1\ \textbf{until}\ \varphi$
  "There is a number N of steps after which φ holds"
- Note that $N = N'' + 1 \leftrightarrow N'' = N - 1 \wedge N > 0$
- Proof of $\Box\, \varphi$ by contradiction:
  Assume a number N of steps after which $\neg\, \varphi$ holds
  Proof is then by well-founded induction over N
- Can be generalized to arbitrary safety properties
  (e.g. sequential programs without local variables)

Induction to prove Fairness

- Weak Fairness: In an interleaving $\alpha \parallel \beta$, program α eventually gets a chance to do a step (if not blocked)
- In TLA: separate formula talking about encoded steps with program counters $\Rightarrow$ not an option of RGITL
- Alternative: General transformation of fair to unfair interleaved programs using counters [Apt,Olderog 91]
- In RGITL: Add an "α is scheduled flag" B:
  $\{B : \alpha \parallel \beta\} \leftrightarrow \{\alpha <\!\parallel \beta\} \vee (\neg \mathrm{B} \wedge \{B : \alpha \parallel\!> \beta\})$
- New Axiom: $\{\alpha \parallel \beta\} \leftrightarrow \exists\, \mathrm{B}.\ \Diamond\, \mathrm{B} \wedge \{B : \alpha \parallel \beta\}$
- $\Diamond$ B allows induction!
- Unfair interleaving satisfies almost the same axiom:
  $$\begin{aligned} \alpha \parallel_{\mathrm{nf}} \beta \equiv \quad & (\exists\, \mathrm{B}.\ \Diamond\, \mathrm{B} \wedge \{B : \alpha \parallel_{\mathrm{nf}} \beta\}) \\ & \vee (\beta \wedge \Box\, (\neg\, \textbf{blocked}) \wedge \mathbf{E}\, \exists\, \underline{x}.\ \alpha) \end{aligned}$$
- $\mathbf{E}\, \exists\, \underline{X}.\ \alpha$: "there is at least one run of α" ($\underline{X}$ = free(α))

Proof principle 3: Compositional Reasoning

- Substitution principle allows to abstract each program in an interleaving to a property
- In particular: Rely/Guarantee rules are expressible
- Guarantee = Predicate for steps of a process G(X,X′)
- Rely = Predicate on environment steps R(X′,X″)
- Program α satisfies R/G, iff:

- As a TL formula: $R \xrightarrow{+} G \equiv \neg\, (R$ **until** $(\neg\, G))$
  (*not* a special operator as in TLA [Lamport 94]!)

Proof principle 3: Compositional Reasoning

- Basic principle:
  - Prove R_i/G_i for interleaved programs α_i (i = 1,2)
  - Prove $G_i \to R_j$ for $i \neq j$, R_i transitive
  - Then: $\alpha_1 \parallel \alpha_2$ satisfies $\Box\, G_1 \vee G_2$
- Provable by using the substitution principle, with
  $A \equiv R_1 \xrightarrow{+} G_1$, $B \equiv R_2 \xrightarrow{+} G_2$, $C \equiv (X' = X'') \xrightarrow{+} G_1 \vee G_2$

$$\frac{\alpha_1 \to A \qquad \alpha_2 \to B \qquad A \parallel B \to C}{\alpha_1 \parallel \alpha_2 \to C}$$

- First two premises = Assumptions for the two programs
- Third premise provable by induction, using
  $R \xrightarrow{+} G \leftrightarrow \forall\, B.\ \Diamond\, B \to (R \wedge \neg\, B) \xrightarrow{+} G$

Rely-Guarantee Theorem

Theorem

(1) $pre \wedge COp_1 \;\rightarrow\; R_1 \xrightarrow{+} (G_1 \wedge (\textbf{last} \rightarrow post_1))$

(2) $pre \wedge COp_2 \;\rightarrow\; R_2 \xrightarrow{+} (G_2 \wedge (\textbf{last} \rightarrow post_2))$

(3) $G_1 \vee R \;\rightarrow\; R_2, G_2 \vee R \;\rightarrow\; R_1, G_1 \vee G_2 \;\rightarrow\; G$

(4) $reflexive(G_1, G_2), transitive(R_1, R_2)$

(5) $pre \wedge (R_1 \vee R_2) \rightarrow pre$

then $pre \wedge COp_1 || COp_2 \;\rightarrow\; R \xrightarrow{+} (G \wedge (\textbf{last} \rightarrow post_1 \wedge post_2))$

- similar to [Xu,deRoever 97] (except cond. (5))
- Their notation for (1): COp_1 $\underline{sat}$ $(pre, rely_1, guar_1, post_1)$
- Deadlock freedom provable too (using **blocked** $\rightarrow$ *wait*)

Outline

- The Logic RGITL
  - Compositional interleaving
  - A semantics with system and environment steps
  - Integration with HOL
- Proof principles of RGITL
  - Symbolic Execution
  - Induction
  - Rely-Guarantee
- Application: Lock-Free Algorithms
  - Motivation
  - Simple Example: Treiber's Stack
  - Linearizability and Lock-Freedom
- Experiences, Future Work

Motivation

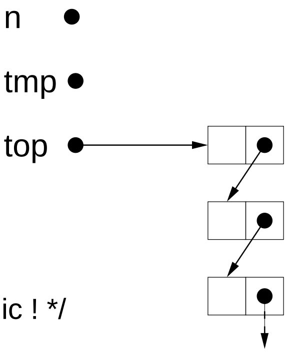
- Multi-core processors getting more and more common
  $\Rightarrow$ Concurrent algorithms more important than ever
- Usually, concurrency is implemented using locks (semaphores, synchronize in Java etc.)
- Lock-free algorithms (also called nonblocking) are an interesting class of algorithms that does not use locks
- Instead they use CAS instructions (x86,Sparc, Itanium) or LL/SC (Alpha,PowerPC)

Example: Treiber's Stack

- Defined in [Treiber 86]
- Implementation of a global stack
- Abstract view: Operations APush and APop
- Implementation with algorithms CPush and CPop
- Representation of stack as a linked list.

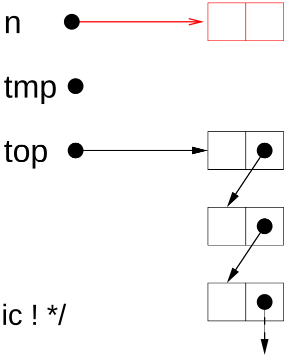
Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}

CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
        then { top := n, success := true; }
    else success := false; }
```



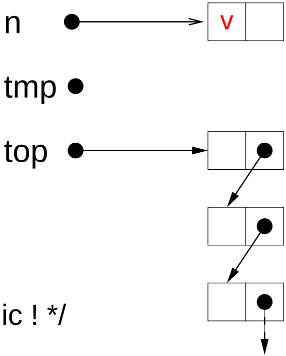
Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}

CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
      then { top := n, success := true; }
    else success := false; }
```



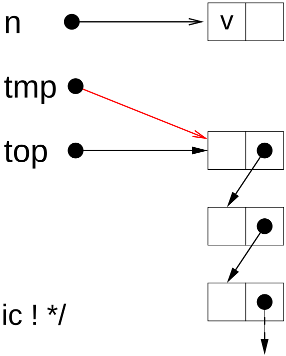
Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}

CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
      then { top := n, success := true; }
    else success := false; }
```



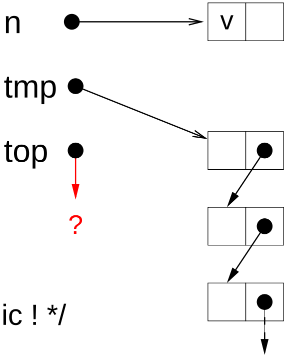
Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}

CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
      then { top := n, success := true; }
    else success := false; }
```



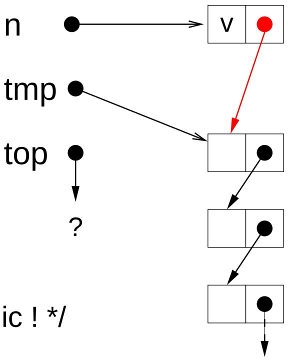
Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}

CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
       then { top := n, success := true; }
    else success := false; }
```



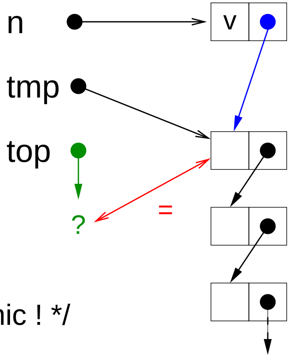
Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}

CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
       then { top := n, success := true; }
    else success := false; }
```



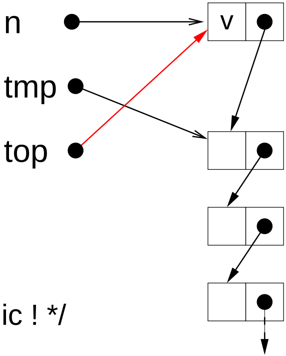
Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}

CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
      then { top := n, success := true; }
    else success := false; }
```



Treiber's Stack - Push

```
CPush(v :Data; top : Pointer) {
    n := new(Node);
    n.val := v;
    sucess := false;
    while sucess = false do {
        tmp := top;
        /* other process .. */
        /* .. may change top! */
        n.next := tmp;
        CAS(tmp, n, top)}
```




```
CAS(tmp, n, top; success) { /* atomic ! */
    if* top = tmp
      then { ; }
    else success := false; }
```

Lock-Free Algorithms and their Use

- Principle of lock-free algorithms:
  - read old data structure
  - prepare modified version
  - update with CAS. Retry on failure
- Treiber's Stack is one of the simplest algorithms (inefficient for high loads; better: [Hendler et. al 04])
- Lock-Free Algorithms exist for many data structures: Queues [Michael, Scott 96], Hashtables [Michael 02], [Gao et al 05], Linked Lists [Harris 01], [Heller 05]
- Used for: process queues, indexes of data bases and Web Servers, real-time 3D games, garbage collection
- Java library supports CAS; f implements lock-free data structures

Locks or no locks?

Advantages of using Locks:

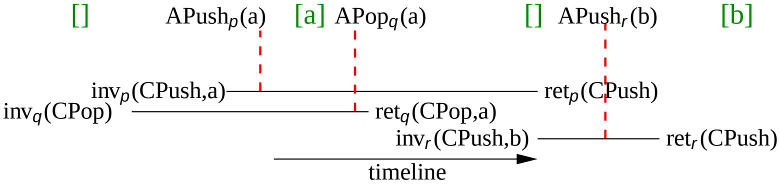
- Well understood, uniform principle
  $\Rightarrow$ easier to verify than lock-free algorithms
  (essentially: verify sequential algorithm)
- Automatic checks for correct use of locks available
- Simple lock-free algorithms are inefficient at high loads: they waste processor time trying over and over

Disadvantages of using Locks:

- Lock is a bottleneck (pessimistic strategy)
- Deadlocks and priority inversion possible
- What happens when the locking process crashes?

Safety: Linearizability

- Defined in [Herlihy & Wing 90]
- Scenario: Several processes (p,q,r), all running algorithm COp in parallel (e.g. CPush $\vee$ CPop)
- Informal definition: Parallel run must be equivalent to a sequential run of AOp (APush $\vee$ APop)



Decomposition of Linearizability

Theorem (Bäumler et al. 09)

If for all $1 \leq p, q \leq n$, $p \neq q$:

(1) $COp_p \rightarrow R_p \xrightarrow{+} G_p$

(2) $G_p \rightarrow R_q$, $reflexive(G_p), transitive(R_p)$, $R \rightarrow R_p$

(3) $COp_p(CS) \wedge \Box\,(R_p \wedge Abs(CS) = AS \wedge Abs(CS') = AS')$
$\rightarrow skip^*; AOp_p(AS); skip^*$

then $COp_1^* \| \ldots \| COp_n^* \wedge \Box\, R \rightarrow AOp_1^* \| \ldots \| AOp_n^* \| skip^*$

- COp_p is a concrete algorithm (procedure) that implements an atomic operation AOp_p
- R is the global environment assumption
- Linearizability expressed as special case of refinement
- Most linearizable algorithms allow reduction to two representative processes $\Rightarrow$ reduction proved

Liveness: Lock-Freedom

For Treiber's Stack:

- CPush may have to retry over and over
  $\Rightarrow$ one single process might be starved
- Every time a retry is necessary, another CPush/CPop must have succeeded and terminated
- This is true, even if the scheduling is unfair, or when a process crashes

Treiber's stack satisfies property of Lock-Freedom:

> *As long as some operations are running, one of them will terminate*

Decomposition of Lock-Freedom

Theorem (Tofan et al. 10)

If for all $0 \leq p, q, p \neq q$:

$$
\begin{array}{ll}
(1) & COp_p \;\rightarrow\; R_p \xrightarrow{+} G_p \\
(2) & G_p \;\rightarrow\; R_q,\ \mathit{reflexive}(G_p), \mathit{transitive}(R_p),\ R \;\rightarrow\; R_p \\
(3) & \mathit{reflexive}(U), \mathit{transitive}(U),\ R \;\rightarrow\; R_p \wedge U \\
(4) & COP_p(CS) \wedge \Box\, R_p \\
 & \rightarrow \Box\, (\neg\, U(CS, CS') \vee (\Box\, U(CS', CS''))\rightarrow \Diamond\, \mathbf{last})
\end{array}
$$

then $COP_0^* \| \ldots \| COP_n^* \wedge \Box\, R \rightarrow \Box\, \mathit{progress}$

where progress = "some operation active $\rightarrow$ *some operation terminates"*

- Predicate U ("unchanged") describes conditions under which $COP_p(CS)$ terminates in environment R_p.
- At any time, COP_p eventually terminates ($\Diamond$ **last**), if:
  - It updates the shared state itself $\neg\, U(CS, CS')$, or
  - It encounters no interference $\Box\, U(CS', CS'')$
- Theorem holds for weak fair and nonfair interleaving

General Experience with the Calculus

- Symbolic execution is natural to verify even concurrent programs:
  - rest of the program directly visible
  - feels much like debugging
- Main new difficulty for proofs is to determine the correct Relys and Guarantees (similar to invariants) in advance
  $\Rightarrow$ Add techniques to automatically infer them
- We've done some significant case studies already:
  Hazard pointers for lock-free algorithms [$\Rightarrow$ tomorrow]
  Medical protocols with synchronous parallel hierarchical plans [Protocure 06]
- Calculus is not yet as easy to use or automated as the wp-calculus for sequential programs (takes time and experience)

Universität Augsburg

Some Open Issues

- Guarantees often hold in a certain section of the code: currently boolean variables must be added manually
  $\Rightarrow$ labels would be helpful, but are incompatible with chop:
  $\alpha;\{L : \beta\}$: final state of α and first of β disagree on L
- express general refinement modulo stuttering
- Prove general commuting diagrams for forward and backward simulation (bounded nondeterminism!)
- Completeness in general is open
  (complete fragments of ITL and RG)

Proving Lock-Free Algorithms

- Calculus is adequate to show correctness of proof obligations (POs) as well as proving instances of the POs for case studies
- Automation is not as high as in related work:
  Automatic checking of linearizability for short operations sequences with model checking [Alur10]
  Automatic proofs for some algorithms using RGSep [Vafeiadis01]
- Nevertheless, the algorithms we check are already more difficult than those that have been proved automatically

Current Work on Lock-Free Algorithms

- Support for Heap modularity is often beneficial
  $\Rightarrow$ develop library with a lightweight embedding of separation
- Open issue: good frame rule for temporal logic?
- Generalize proof obligations (POs) for linearizability
  (POs shown require lin. points within executing thread
  $\Rightarrow$ complete POs for arbitrary lin. points

Major Challenge:
Interleaving assumes sequentially consistent memory, but:
Processors use weak memory models
(and Javas much debated memory model is even weaker)