First Year Report

Loïc Fejoz

March 2, 2007

Contents

1 Introduction

One year ago my project proposal was to apply well-known methods in my research team to lock-free algorithms. Those methods are B, TLA^+ and model-checking. Those methods imply a top-down approach. So the idea was to develop new algorithms. Since then I discovered a lot of algorithms and a lot of work on developing such algorithms. But proofs were very rare and none of them were written within a proof-assistant. Therefore, I want to justify lock-free algorithms by applying well-known methods, and in particular B and TLA^+.

In this report, I will first introduce lock-free algorithms and their properties. I will also present methods that should be useful for proving algorithms. Then I will expose my preliminary work with those methods. I will end with suggestions for an easier method to use for proving the correctness of lock-free algorithms.

2 Related work

2.1 Lock-free libraries and algorithms

Usually algorithms that manage a shared data structure use locks. Lock-free algorithms are designed such that they allow multiple processes (a.k.a. threads) to access the shared data structure concurrently, either for reading or writing. Instead of central locking, these algorithms rely on particular atomic primitives. It solve problems like deadlock and priority inversion. Depending on the contention it can also be more efficient. Those new algorithms are possible because hardwares now provide some complex atomic primitives.

A lot of work as been done in the last three years. It starts from research articles about algorithms (with justifications) [16, 7, 17, 12], to fully useable libraries [3, 12, 1]. In between we can find stand-alone algorithms [11, 2, 4].

Another interesting project is Software Transaction Memory [5]. It is partially related to lock-free algorithms. Indeed it provides programmers with a high-level view of transaction. In the background (implementation) it is based on lock-free algorithms. It would be nice for the target method to be able to prove those implementations.

2.2 Basic notions

Actually a data-structure is not changed by a single algorithm but by several. For instance, on a list you will find an algorithm for appending an element, one for retrieving while deleting elements, and one for inserting. Those algorithms can be run concurrently by several process.

When reasoning about a data-structure, it helps if all methods appear to be executed atomically. This property is called linearisability [10]. A concurrent execution trace is linearisable iff there exists a sequential execution trace that has the same externally observable behaviour.

```
x = f(x);
```

Figure 1: A non-implementable assignment

Usually locks are replaced by loops that use atomic primitives provided by the processor (like the CAS_1 presented in section 3.1). So we can wonder about termination. This leads to the following definition. I have not focused yet on liveness properties and termination problem.

wait-free All threads make progress even if others incur delays;

lock-free Some thread always makes progress;

obstruction-free guarantees progress for any thread that eventually executes in isolation.

3 My Work

3.1 Introduction

During the first year of the thesis, my work was around the use of Compare-and-Swap (a.k.a. CAS). CAS_1 is an atomic operation that can be found on modern processors. It is also known as a universal constructor for atomic operations [9]. CAS_1 is equivalent to the atomic execution of:

```
word_t CAS1(word_t *a, word_t o, word_t n) {
  old = *a;
  if (old == o) {
    *a = n;
  }
  return old;
}
```

I will sometimes use an operation called $BCAS_1$. It is similar to CAS_1 but returns TRUE if the reference cell was updated and FALSE otherwise.

In this section I will present my preliminary experiences with proving a very simple concurrent algorithm. But one must keep in mind that my goal is to prove algorithms from [8]. There is a set of patterns used by Lock-free algorithms. This is were the following pattern comes from.

The pattern is simply replacing one assignment of the type shown in figure 1 by the program shown in figure 2, where f is a pure function (*i.e.* it has no side effects).

I want to prove that the latter could substitute the former. But to do so I need to tell in which environment the algorithm is used. Following the

```
b = false;
do {
  v = x;
  fv = f(v);
  b = BCAS1(x, v, fv);
} while(!b);
```

Figure 2: Assignment with $BCAS_1$

assume-guarantee method, I want to compose the algorithm with another process that can assign any value (respecting the type) to x.

In the following sections I will explain my attempts to prove this algorithm in different languages. Some of them are supported by tools.

3.2 Assume-Guarantee in Isabelle

A rely-guarantee tuple $\langle rely, guarantee\rangle\{\phi\}P\{\psi\}$ is pretty much like a Hoare triple but it adds two predicates: *rely* that expresses what this program P relies on from the other programs to be correct, and *guarantee* that expresses what this part P guarantees to the other. This logic [14] has been encoded in Isabelle [15].

One part of proving the assignment example is to show that the tuple $\langle b' = b \wedge fv' = fv \wedge v' = v, x' = x \vee x' = f(x)\rangle\{\top\}P\{\top\}$ holds (P being the program on figure 2). The first step was to modify the language to add $BCAS_1$ in the primitive language (defined in appendix A). The proof is given in appendix A.

The proof is only a partial one as we have not proved that the assignment is done only once. For that purpose we could have added some ghost variable that counts how many time we are doing the assignment. The post-condition would then assert that the counter equals one.

This framework does not suit well our purpose because:

1. you need extra variables;

2. it does not permit to develop a family of algorithms;

3. the proof is messed up with encoding details;

3.3 B method

The simplest way to reason about concurrent program is by using invariants. It is one of the main ideas of the B method besides refinement. The B method was elaborated by Jean-Raymond Abrial [6]. It is a well-established method and several tools exist to prove refinements. Indeed the method describes a system by a set of events. Then you can refine a specification by either adding concrete events or by adding data refinement.

Appendix A presents a simple B machine M0 that just defines a system where two kinds of events can happen, either the guarantee or the rely event. Then we derive two refinement machines. The first one, M1 (on page 8), explicitly introduces the concrete operations, and the second one, M2 (on page 8), adds flow control with a place predicate.

The B method can help us to co-develop several algorithms but it does not ease the reading of the invariant by not allowing local invariants. Also refinement needs to be done on a per event basis, that is why in the most abstract specification M0 (on page 6), the action *guar* appears several times. Making several (clever) refinements helps the proofs to be done automatically; otherwise interactive proofs would have been necessary.

3.4 TLA^+ and ^+CAL

I wrote similar specifications as the previous ones but with TLA^+. Unfortunately, TLA^+ only comes with a model-checker. But Lamport has defined an interesting language called ^+CAL [13]. ^+CAL is an algorithm language. It is meant to replace pseudo-code for writing high-level descriptions of algorithms. ^+CAL specifications are then translated into TLA^+.

^+CAL and TLA^+ could have been well-suited for our problem provided that you could choose the granularity of statements and that a prover existed. The notation is powerful and elegant. Also one good point is that you do not need to express which event refines which one, *i.e.* it does not ask for a linearisability point as B does.

4 Future work and conclusion

It was very insightful and interesting to try to prove such a simple algorithm with those different methods. It let me discover their strengths and weaknesses towards our problem.

So a well-suited method for our problem should allow

- to jointly develop several algorithms,
- to refine from an atomic specification to a lock-free one,
- to compose several specifications,
- to express linearisability points only when needed,
- one algorithm to do job in advance for another one — another common pattern in lock-free algorithms —,
- not to use explicit place predicates.

I have recently started to define such a method and I will further develop it in the comming months. I will then use it to prive at least the RDCSS algorithm.

5 Courses

5.1 TiC06

In July, I attended the summer school TiC. TiC stands for Trends in Concurrency. The goal of the school was to expose graduate students and young researchers to new ideas in concurrent programming from experts in academia and industry. The school was organised at the Centro Residenziale Universitario of the University of Bologna, situated in Bertinoro. More details can be found at http://www.cs.purdue.edu/homes/jv/events/TiC06/.

Several courses where directly useful for my subject. The most notable one was "Highly Concurrent Data Structures" by Maurice Herlihy. Indeed he presented an algorithm for set that uses a lock-free list. It would be nice to prove its correctness. There were also courses about memory models and process algebras. It was also interesting to meet people to whom state your problem, do some tools demo and get some tips and ideas.

5.2 Models and multi-agents system

This course was part of my local obligation from the PhD program in Nancy. It is not strictly related to my topics as it presented concepts for multi-agents system and their building. It then focused on reactive multi-agents for simulating complex phenomena. Finally it presented a cognitive model of agents. More details can be found at http://fst.uhp-nancy.fr/details/form_ue/form_ue_MAIF3U22.html. But it was interesting to consider their problems, especially how to link (and prove?) the global system behaviour towards the local agent's behaviour.

A B Machines

```
MODEL
  M0
/* This is a simple modelisation of the following RG Tuple:
<b:=b & v:=v & fv:=fv & x:: INTEGER, x:=f(x) or x:=x>{TRUE}x:=f(x){TRUE} */
VARIABLES
  x, b, fv, v
CONSTANTS
  f
PROPERTIES
  f : INTEGER --> INTEGER
INVARIANT
```

```
  x : INTEGER &
  b : BOOL &
  v : INTEGER &
  fv : INTEGER
INITIALISATION
  x :: INTEGER ||
  b :: BOOL ||
  v :: INTEGER ||
  fv :: INTEGER
EVENTS
/* This algorithm rely on the other to not change his local
variables but they can modify x */
  rely = BEGIN
      b := b ||
      v := v ||
      fv := fv ||
      x :: INTEGER
  END;

  guar1a = BEGIN
      x := x ||
      b :: BOOL ||
      v :: INTEGER ||
      fv :: INTEGER
  END;

  guar1b = BEGIN
      x := x ||
      b :: BOOL ||
      v :: INTEGER ||
      fv :: INTEGER
  END;

  guar1c = BEGIN
      x := x ||
      b :: BOOL ||
      v :: INTEGER ||
      fv :: INTEGER
  END;

  guar1d = BEGIN
      x := x ||
      b :: BOOL ||
      v :: INTEGER ||
      fv :: INTEGER
  END;

/* This is the real event that does the assignment */
  guar2d = BEGIN
      x := f(x) ||
      b :: BOOL ||
      v :: INTEGER ||
      fv :: INTEGER
  END
```

```
END
```

```
REFINEMENT
   M1
/* Here w begin to replace the guarante by
what the algorithm will do but without any order yet.*/
REFINES
   M0
VARIABLES
   x, b, fv, v
INITIALISATION
   x :: INTEGER ||
   b :: BOOL ||
   v :: INTEGER ||
   fv :: INTEGER
EVENTS
   guar1a = BEGIN
         b := FALSE
   END;

   guar1b = BEGIN
         v := x
   END;

   guar1c = BEGIN
         fv := f(v)
   END;

   guar1d = SELECT (x /= v) THEN
         x := x ||
         b := FALSE
   END;

/* The test x=v if the one that will be done by the CAS operation.
The second condition is necessary yet but will be remove in the next refinement.
It correspond to a local invariant in the original algorithm. */
   guar2d = SELECT x = v & fv = f(v)THEN
         x := fv ||
         b := TRUE
   END
END
```

```
REFINEMENT
   M2
/* This is the concrete algorithm. We must have added the place
predicate pc (aka program counter) and the correspondant
labels lbl.
We have then prove that
x := f(v)
can be replace by
```

```
b := FALSE;
do {
  v := x;
  fv := f(v);
  b := BCAS_1(x, v, fv)
} while (not b)
*/
REFINES
   M1
SETS
   lbl = {PCA, PCB, PCC, PCD, PCE}
VARIABLES
   x, b, fv, v, pc
INVARIANT
   pc : lbl &
   ((pc = PCD) => (fv = f(v))) /* Here is the local invariant that was placed on the guard in the previous refinem
INITIALISATION
   x :: INTEGER ||
   b :: BOOL ||
   v :: INTEGER ||
   fv :: INTEGER ||
   pc := PCA
EVENTS
   guar1a = SELECT pc = PCA THEN
        b := FALSE ||
        pc := PCB
   END;

   guar1b = SELECT pc = PCB THEN
        v := x ||
        pc := PCC
   END;

   guar1c = SELECT pc = PCC THEN
        fv := f(v) ||
        pc := PCD
   END;

/* In that case the CAS test fails. b := CAS_1(x, v, fv)*/
   guar1d = SELECT pc = PCD & x /= v THEN
        x := x ||
        b := FALSE ||
        pc := PCB
   END;

/* The CAS operation is done. b := BCAS_1(x, v, fv)*/
   guar2d = SELECT pc = PCD & x = v THEN
        x := fv ||
        b := TRUE ||
        pc := PCE
   END

END
```

B Concrete Syntax

```
theory RG_Syntax
imports "~~/src/HOL/HoareParallel/RG_Hoare" "~~/src/HOL/HoareParallel/Quote_Antiquote"
begin

syntax
  "_Assign"    :: "idt ⇒ 'b ⇒ 'a com"                        ("(´_ :=/
_)" [70, 65] 61)
  "_skip"      :: "'a com"                                   ("SKIP")
  "_Seq"       :: "'a com ⇒ 'a com ⇒ 'a com"                 ("(_;;/ _)"
[60,61] 60)
  "_Cond"      :: "'a bexp ⇒ 'a com ⇒ 'a com ⇒ 'a com"   ("(0IF _/
THEN _/ ELSE _/FI)" [0, 0, 0] 61)
  "_Cond2"     :: "'a bexp ⇒ 'a com ⇒ 'a com"             ("(0IF _ THEN
_ FI)" [0,0] 56)
  "_While"     :: "'a bexp ⇒ 'a com ⇒ 'a com"             ("(0WHILE
_ /DO _ /OD)"  [0, 0] 61)
  "_Await"     :: "'a bexp ⇒ 'a com ⇒ 'a com"             ("(0AWAIT
_ /THEN /_ /END)"  [0,0] 61)
  "_Atom"      :: "'a com ⇒ 'a com"                          ("(⟨_⟩)" 61)
  "_Wait"      :: "'a bexp ⇒ 'a com"                         ("(0WAIT _
END)" 61)
  "_Cas"       :: "idt ⇒ 'b ⇒ 'b ⇒ 'a com"                    ("(CAS ´_,
_, _ SAC)" [70, 65, 65] 61)
  "_Cas2"      :: "idt ⇒ idt ⇒ 'b ⇒ 'b ⇒ 'a com"            ("(´_ :=
CAS ´_, _, _ SAC)" [71, 71, 65, 65] 61)
  "_CasB"      :: "idt ⇒ idt ⇒ 'b ⇒ 'b ⇒ 'a com"            ("(´_ :=
BCAS ´_, _, _ SACB)" [71, 71, 65, 65] 61)

translations
  "´  x := a" ⇀ "Basic «´  (_update_name x a)»"
  "SKIP" ⇌ "Basic id"
  "c1;; c2" ⇌ "Seq c1 c2"
  "IF b THEN c1 ELSE c2 FI" ⇀ "Cond .{b}. c1 c2"
  "IF b THEN c FI" ⇌ "IF b THEN c ELSE SKIP FI"
  "WHILE b DO c OD" ⇀ "While .{b}. c"
  "AWAIT b THEN c END" ⇌ "Await .{b}. c"
  "⟨c⟩" ⇌ "AWAIT True THEN c END"
  "WAIT b END" ⇌ "AWAIT b THEN SKIP END"
  "CAS ´x, a, n SAC" ⇀ "⟨IF ´x=a THEN ´x:=n FI⟩"
  "´r := CAS ´x, a, n SAC" ⇀ "⟨´r:=´x;; (IF ´x=a THEN ´x:=n FI)⟩"
  "´r := BCAS ´x, a, n SACB" ⇀ "⟨(IF ´x=a THEN ´x:=n;; ´r:=(1::nat)
ELSE ´r:=(0::nat) FI)⟩"


nonterminals
  prgs
```

syntax
```
  "_PAR"        :: "prgs ⇒ 'a"              ("COBEGIN//_//COEND" 60)
  "_prg"        :: "'a ⇒ prgs"              ("_" 57)
  "_prgs"       :: "['a, prgs] ⇒ prgs"      ("_//‖//_" [60,57] 57)
```

translations
```
  "_prg a" ⇀ "[a]"
  "_prgs a ps" ⇀ "a # ps"
  "_PAR ps" ⇀ "ps"
```

syntax
```
  "_prg_scheme" :: "['a, 'a, 'a, 'a] ⇒ prgs"  ("SCHEME [_ ≤ _ < _] _"
[0,0,0,60] 57)
```

translations
```
  "_prg_scheme j i k c" ⇌ "(map (λi. c) [j..<k])"
```

Translations for variables before and after a transition:

syntax
```
  "_before" :: "id ⇒ 'a" ("ᵒ_")
  "_after"  :: "id ⇒ 'a" ("ᵃ_")
```

translations
```
  "ᵒx" ⇌ "x ´fst"
  "ᵃx" ⇌ "x ´snd"
```

print_translation {*
```
  let
    fun quote_tr' f (t :: ts) =
          Term.list_comb (f $ Syntax.quote_tr' "_antiquote" t, ts)
      | quote_tr' _ _ = raise Match;

    val assert_tr' = quote_tr' (Syntax.const "_Assert");

    fun bexp_tr' name ((Const ("Collect", _) $ t) :: ts) =
          quote_tr' (Syntax.const name) (t :: ts)
      | bexp_tr' _ _ = raise Match;

    fun upd_tr' (x_upd, T) =
      (case try (unsuffix RecordPackage.updateN) x_upd of
        SOME x => (x, if T = dummyT then T else Term.domain_type T)
      | NONE => raise Match);

    fun update_name_tr' (Free x) = Free (upd_tr' x)
      | update_name_tr' ((c as Const ("_free", _)) $ Free x) =
          c $ Free (upd_tr' x)
      | update_name_tr' (Const x) = Const (upd_tr' x)
      | update_name_tr' _ = raise Match;
```

```
    fun assign_tr' (Abs (x, _, f $ t $ Bound 0) :: ts) =
          quote_tr' (Syntax.const "_Assign" $ update_name_tr' f)
            (Abs (x, dummyT, t) :: ts)
      | assign_tr' _ = raise Match;
  in
    [("Collect", assert_tr'), ("Basic", assign_tr'),
      ("Cond", bexp_tr' "_Cond"), ("While", bexp_tr' "_While_inv")]
  end
*}
```

end

C Assignment

theory *RG_Assign* **imports** *RG_Syntax* **begin**

lemmas *definitions [simp]= stable_def Pre_def Rely_def Guar_def Post_def Com_def*

C.1 Atomic assignement

```
record AtomicAssign =
  x  :: nat

lemma AtomicAssign:
shows "⊢ ´x := f ´x sat [⦃ True ⦄ ,⦃ True ⦄ ,⦃ªx=f ºx ∨ ªx=ºx ⦄ ,⦃
True ⦄ ]"
proof (rule Basic)
  show "⦃True⦄ ⊆ ⦃´(AtomicAssign.x_update (f ´AtomicAssign.x)) ∈ ⦃True⦄⦄"
    by auto
  next
    show "stable ⦃True⦄ ⦃True⦄"
      by auto
  next
    show "stable ⦃True⦄ ⦃True⦄"
      by auto
  next
    show "{(s, t). s ∈ ⦃True⦄ ∧ (t = s⦇AtomicAssign.x := f (AtomicAssign.x
s)⦈ ∨ t = s)} ⊆ ⦃ªx = f ºx ∨ ªx = ºx⦄"
      by (simp, auto)
qed
```

C.2 Atomic assignement with CAS

```
record AssignWithCAS =
  x  :: nat
  v  :: nat
  fv :: nat
  b  :: nat
```

```
lemma AssignWithCAS:
  shows "⊢ ´b := (0::nat);; WHILE ´b = (0::nat) DO ´v := ´x;; ´fv :=
f ´v;; ´b := BCAS ´x, ´v, ´fv SACB OD sat [⦃ True ⦄ ,⦃ªv=ºv ∧ ªb=ºb
∧ ªfv=ºfv⦄ ,⦃ªx=ºx ∨ ªx=f ºx⦄ ,⦃ True ⦄ ]"
proof (rule_tac mid="⦃ ´b = 0 ⦄" in Seq)
  show "⊢ ´b := 0 sat [⦃ True ⦄ ,⦃ªv=ºv ∧ ªb=ºb ∧ ªfv=ºfv⦄ ,⦃ªx=ºx
∨ ªx=f ºx ⦄ ,⦃ ´b = 0 ⦄ ]"
    by (rule Basic, auto)
next
  show "⊢ WHILE ´b = (0::nat) DO ´v := ´x;; ´fv := f ´v;; ´b := BCAS
´x, ´v, ´fv SACB OD sat [⦃ ´b = 0 ⦄ ,⦃ªv=ºv ∧ ªb=ºb ∧ ªfv=ºfv⦄ ,⦃ªx=ºx
∨ ªx=f ºx⦄ ,⦃ True ⦄]"
  proof (rule_tac pre'="⦃ True ⦄" and
      guar'="⦃ ªx=ºx ∨ ªx=f ºx ⦄" and
      rely'="⦃ ªv=ºv ∧ ªb=ºb ∧ ªfv=ºfv ⦄" and
      post'="⦃ True ⦄" in Conseq)
    show "⦃ ´b = 0 ⦄ ⊆ ⦃ True ⦄"
      by auto
  next
    show "⦃ªv = ºv ∧ ªb = ºb ∧ ªfv = ºfv⦄ ⊆ ⦃ªv = ºv ∧ ªb = ºb ∧ ªfv
= ºfv⦄"
      by auto
  next
    show "⦃ªx = ºx ∨ ªx = f ºx⦄ ⊆ ⦃ªx = ºx ∨ ªx = f ºx⦄"
      by auto
  next
    show "⦃True⦄ ⊆ ⦃True⦄"
      by auto
  next
    show "⊢ WHILE ´b = 0 DO ´v := ´x;; ´fv := f ´v;; ´b := BCAS ´x,
´v, ´fv SACB OD sat [⦃ True ⦄ ,⦃ªv=ºv ∧ ªb=ºb ∧ ªfv=ºfv⦄ ,⦃ªx=ºx ∨ ªx=f
ºx⦄ ,⦃ True ⦄]"
    proof (rule While)
      show "stable ⦃True⦄ ⦃ªv = ºv ∧ ªb = ºb ∧ ªfv = ºfv⦄"
        by auto
    next
      show "⦃True⦄ ∩ - ⦃´b = 0⦄ ⊆ ⦃True⦄"
        by auto
    next
      show "stable ⦃True⦄ ⦃ªv = ºv ∧ ªb = ºb ∧ ªfv = ºfv⦄"
        by auto
    next
      show "∀s. (s, s) ∈ ⦃ªx = ºx ∨ ªx = f ºx⦄"
        by auto
    next
      show "⊢ ´v := ´x;; ´fv := f ´v;; ⟨IF ´x = ´v THEN ´x := ´fv;;
´b := 1 ELSE ´b := 0 FI⟩ sat [⦃True⦄ ∩ ⦃´b = 0⦄, ⦃ªv = ºv ∧ ªb = ºb
```

```
∧ ªfv = ºfv⦄, ⦃ªx = ºx ∨ ªx = f ºx⦄, ⦃True⦄]"
      proof (rule_tac pre'="⦃ ´b = 0 ⦄" and
          guar'="⦃ ªx=ºx ∨ ªx=f ºx ⦄" and
          rely'="⦃ ªv=ºv ∧ ªb=ºb ∧ ªfv=ºfv ⦄" and
          post'="⦃ True ⦄" in Conseq)
        show "⦃True⦄ ∩ ⦃´b = 0⦄ ⊆ ⦃´b = 0⦄"
          by auto
      next
        show "⦃ªv = ºv ∧ ªb = ºb ∧ ªfv = ºfv⦄ ⊆ ⦃ªv = ºv ∧ ªb = ºb
∧ ªfv = ºfv⦄"
          by auto
      next
        show "⦃ªx = ºx ∨ ªx = f ºx⦄ ⊆ ⦃ªx = ºx ∨ ªx = f ºx⦄"
          by auto
      next
        show "⦃True⦄ ⊆ ⦃True⦄"
          by auto
      next
        show "⊢ ´v := ´x;; ´fv := f ´v;; ⟨IF ´x = ´v THEN ´x := ´fv;;
´b := 1 ELSE ´b := 0 FI⟩ sat [⦃´b = 0⦄, ⦃ªv = ºv ∧  ªb = ºb ∧ ªfv = ºfv⦄,
⦃ªx = ºx ∨ ªx = f ºx⦄, ⦃True⦄]"
        proof (rule_tac mid="⦃ ´fv = f ´v  ⦄" in Seq)
          show "⊢ ´v := ´x;; ´fv := f ´v sat [⦃´b = 0⦄, ⦃ªv = ºv ∧
ªb = ºb ∧ ªfv = ºfv⦄, ⦃ªx = ºx ∨ ªx = f ºx⦄, ⦃ ´fv = f ´v ⦄]"
          proof (rule_tac mid="⦃True⦄" in Seq)
            show "⊢ ´v := ´x sat [⦃´b = 0⦄, ⦃ªv = ºv ∧  ªb = ºb ∧ ªfv
= ºfv⦄, ⦃ªx = ºx ∨ ªx = f ºx⦄, ⦃ True ⦄]"
              by (rule Basic, auto)
          next
            show "⊢ ´fv := f ´v sat [⦃True⦄, ⦃ªv = ºv ∧  ªb = ºb ∧
ªfv = ºfv⦄, ⦃ªx = ºx ∨ ªx = f ºx⦄, ⦃ ´fv = f ´v ⦄]"
              by (rule Basic, auto)
          qed
        next
          show "⊢ ⟨IF ´x = ´v THEN ´x := ´fv;; ´b := 1 ELSE ´b := 0
FI⟩ sat [⦃´fv = f ´v ⦄, ⦃ªv = ºv ∧  ªb = ºb ∧ ªfv = ºfv⦄, ⦃ªx = ºx ∨
ªx = f ºx⦄, ⦃True⦄]"
          proof (rule Await,auto)
            show "⋀V. ⊢ IF ´x = ´v THEN ´x := ´fv;; ´b := Suc 0 ELSE
´b := 0 FI sat [⦃´fv = f ´v⦄ ∩ {V}, {(s, t). s = t}, UNIV, ⦃´x = x V
∨ ´x = f (x V)⦄]"
            proof (rule Cond, auto)
              show "⋀V. ⊢ ´x := ´fv;; ´b := Suc 0 sat [⦃´fv = f ´v⦄
∩ {V} ∩  ⦃´x = ´v⦄, {(s, t). s = t}, UNIV, ⦃´x = x V ∨ ´x = f (x V)⦄]"
              proof (rule_tac mid="⦃´x = x V ∨ ´x = f (x V)⦄" in Seq)
                show "⋀V. ⊢ ´x := ´fv sat [⦃´fv = f ´v⦄ ∩ {V} ∩ ⦃´x
= ´v⦄, {(s, t). s = t}, UNIV,⦃´x = x V ∨ ´x = f (x V)⦄ ]"
                  by (rule Basic, auto)
              next
```

```
                show "⋀V. ⊢ ´b := Suc 0 sat [⦃´x = x V ∨ ´x = f (x
V)⦄, {(s, t). s = t}, UNIV, ⦃´x = x V ∨ ´x = f (x V)⦄]"
                  by (rule Basic, auto)
              qed
            next
              show " ⋀V. ⊢ ´b := 0 sat [⦃´fv = f ´v⦄ ∩ {V} ∩   - ⦃´x
= ´v⦄, {(s, t). s = t}, UNIV, ⦃´x = x V ∨ ´x = f (x V)⦄]"
                by (rule Basic, auto)
            qed
          qed
        qed
      qed
    qed
  qed
qed

end
```

D Bibliography

References

[1] Atomic Ptr Plus. Available from World Wide Web: http://atomic-ptr-plus.sourceforge.net/. This project is a collection of various lock-free synchronization primitives and fast pathed synchronization functions.

[2] Generic Concurrent Lock-Free Linked List. Available from World Wide Web: http://www.cs.rpi.edu/~bushl2/project_web/page5.html.

[3] NOBLE. Available from World Wide Web: http://www.cs.chalmers.se/~noble/.

[4] Nonblocking multiprocessor/multithread algorithms in C++. Available from World Wide Web: http://www.musicdsp.org/archive.php?classid=0#148.

[5] Software Transactional Memory. Available from World Wide Web: http://en.wikipedia.org/wiki/Software_transactional_memory.

[6] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[7] H. Gao. *Design and verification of lock-free parallel algorithms*. PhD thesis, Apr. 2005. Available from World Wide Web: http://dissertations.ub.rug.nl/faculties/science/2005/h.gao/.

[8] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002. Available from World Wide Web: http://www.cl.cam.ac.uk/research/srg/netos/lock-free/.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991. Available from World Wide Web: http://citeseer.ist.psu.edu/herlihy93waitfree.html.

[10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. Available from World Wide Web: http://doi.acm.org/10.1145/78969.78972.

[11] HP. Atomic Ops. Available from World Wide Web: http://www.hpl.hp.com/research/linux/atomic_ops/.

[12] Keir Fraser, Tim Harris, Ian Pratt, and Chris Purcell. Practical lock-free data structures. Available from World Wide Web: http://www.cl.cam.ac.uk/research/srg/netos/lock-free/.

[13] L. Lamport. The ^{+}CAL Algorithm Language. Available from World Wide Web: http://research.microsoft.com/users/lamport/pubs/pubs.html#pluscal.

[14] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. Available from World Wide Web: http://portal.acm.org/citation.cfm?id=360051.360224.

[15] L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002. Available from World Wide Web: http://tumb1.biblio.tu-muenchen.de/publ/diss/allgemein.html.

[16] H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. Technical Report 2004-02, Computing Science, Chalmers University of Technology, Mar. 2004. Available from World Wide Web: http://citeseer.ist.psu.edu/sundell04lockfree.html.

[17] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice*

of parallel programming, pages 129–136, New York, NY, USA, 2006. ACM Press. Available from World Wide Web: http://doi.acm.org/10.1145/1122971.1122992.