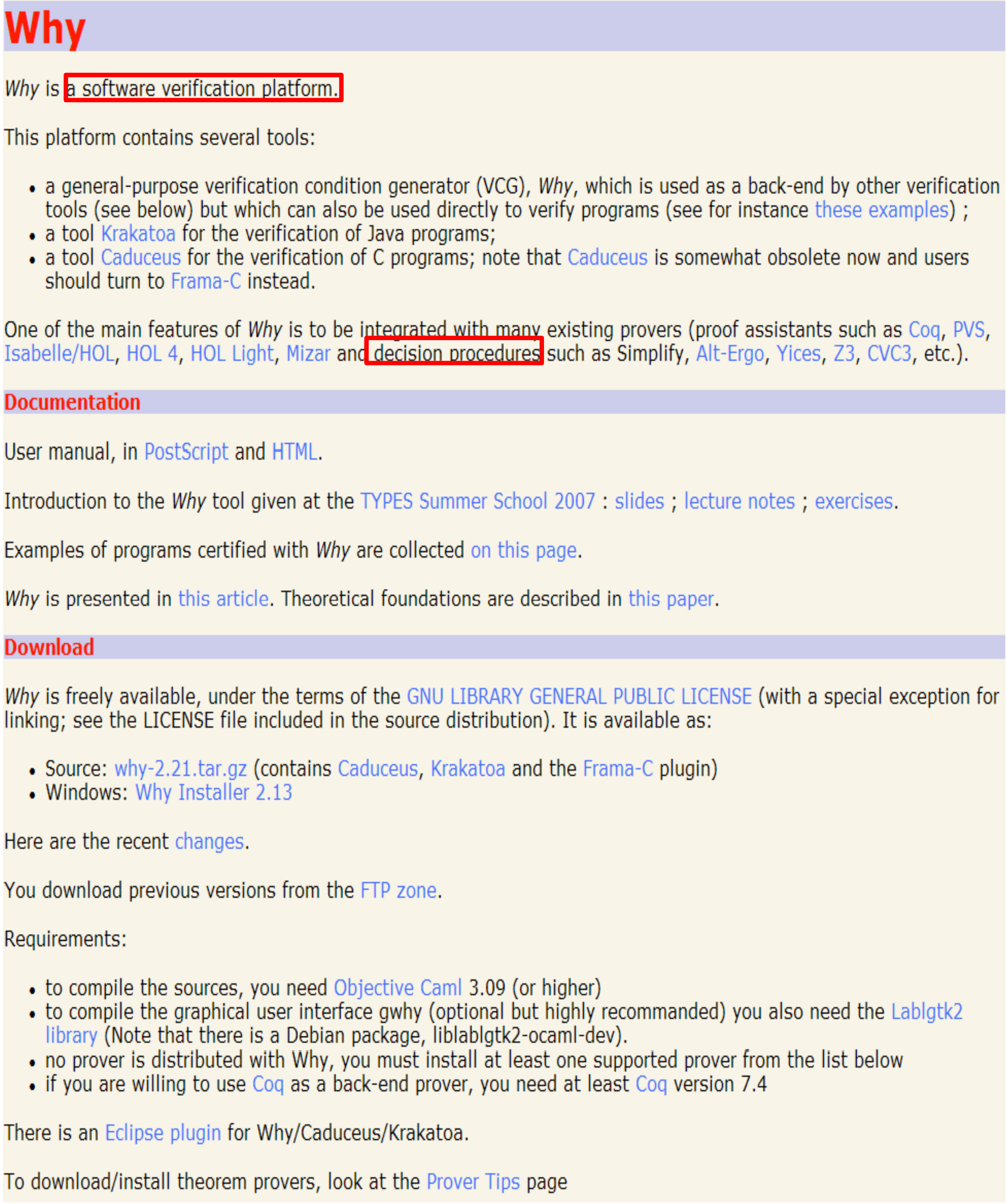
WHY Tutorial

Moonzoo Kim

Provable Software Laboratory

CS Dept. KAIST



Why

Why is a software verification platform.

This platform contains several tools:

- a general-purpose verification condition generator (VCG), *Why*, which is used as a back-end by other verification tools (see below) but which can also be used directly to verify programs (see for instance these examples) ;
- a tool Krakatoa for the verification of Java programs;
- a tool Caduceus for the verification of C programs; note that Caduceus is somewhat obsolete now and users should turn to Frama-C instead.

One of the main features of *Why* is to be integrated with many existing provers (proof assistants such as Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar and decision procedures such as Simplify, Alt-Ergo, Yices, Z3, CVC3, etc.).

Documentation

User manual, in PostScript and HTML.

Introduction to the *Why* tool given at the TYPES Summer School 2007 : slides ; lecture notes ; exercises.

Examples of programs certified with *Why* are collected on this page.

Why is presented in this article. Theoretical foundations are described in this paper.

Download

Why is freely available, under the terms of the GNU LIBRARY GENERAL PUBLIC LICENSE (with a special exception for linking; see the LICENSE file included in the source distribution). It is available as:

- Source: why-2.21.tar.gz (contains Caduceus, Krakatoa and the Frama-C plugin)
- Windows: Why Installer 2.13

Here are the recent changes.

You download previous versions from the FTP zone.

Requirements:

- to compile the sources, you need Objective Caml 3.09 (or higher)
- to compile the graphical user interface gwhy (optional but highly recommanded) you also need the Lablgtk2 library (Note that there is a Debian package, liblablgtk2-ocaml-dev).
- no prover is distributed with Why, you must install at least one supported prover from the list below
- if you are willing to use Coq as a back-end prover, you need at least Coq version 7.4

There is an Eclipse plugin for Why/Caduceus/Krakatoa.

To download/install theorem provers, look at the Prover Tips page

Motivating Example

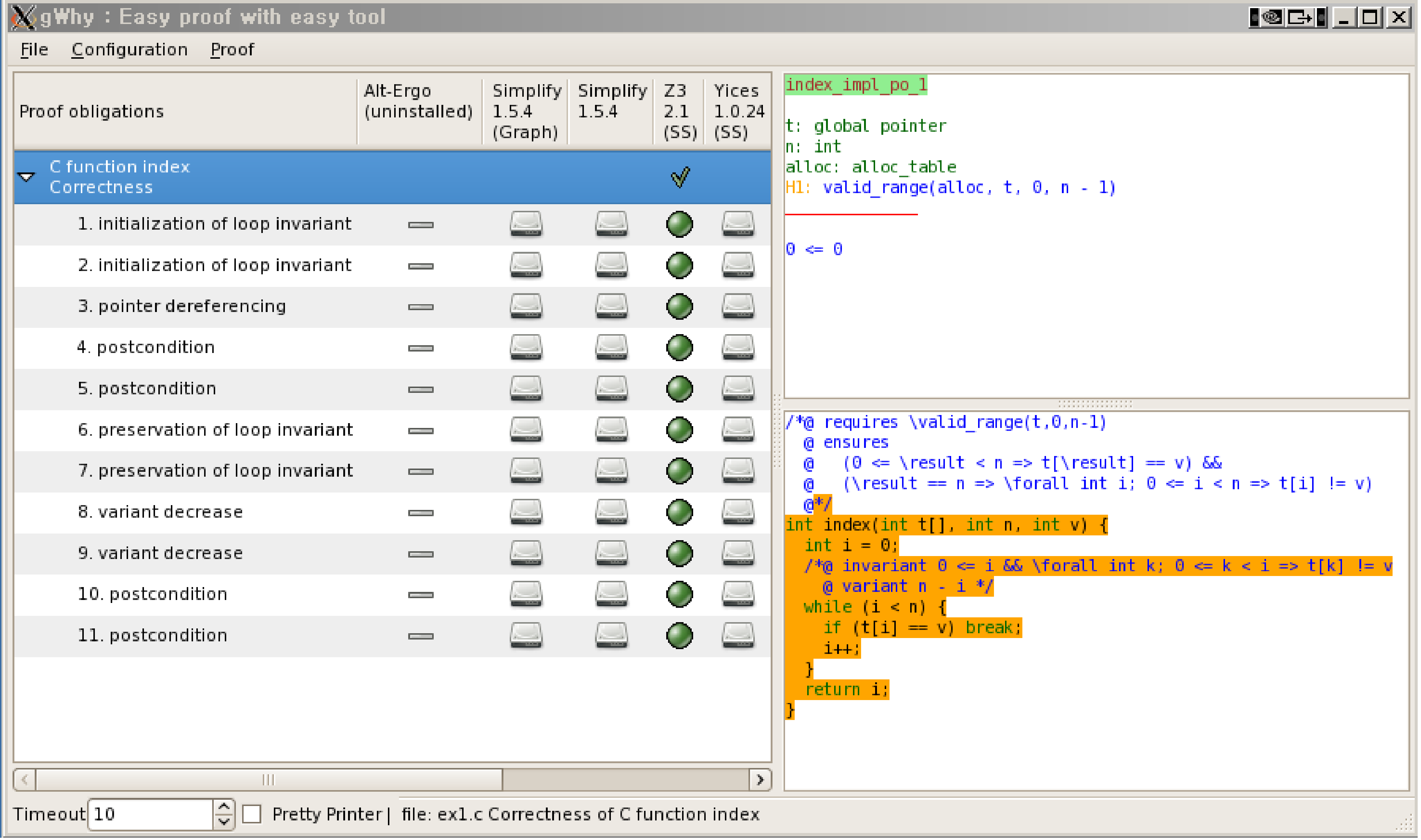
```
/*@ requires ₩valid_range(t,0,n-1)
  @ ensures
  @   (0 <= ₩result < n => t[₩result] == v) &&
  @   (₩result == n => ₩forall int i; 0 <= i < n => t[i] != v)
  @*/
int index(int t[], int n, int v) {
  int i = 0;
  /*@ invariant 0 <= i && ₩forall int k; 0 <= k < i => t[k] != v
    @ variant n - i */
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

dereferenceable

behavioral correctness

termination

Snapshot of GUI of WHY



Proof obligations	Alt-Ergo (uninstalled)	Simplify 1.5.4 (Graph)	Simplify 1.5.4	Z3 2.1 (SS)	Yices 1.0.24 (SS)
C function index Correctness				✓	
1. initialization of loop invariant	—	▭	▭	●	▭
2. initialization of loop invariant	—	▭	▭	●	▭
3. pointer dereferencing	—	▭	▭	●	▭
4. postcondition	—	▭	▭	●	▭
5. postcondition	—	▭	▭	●	▭
6. preservation of loop invariant	—	▭	▭	●	▭
7. preservation of loop invariant	—	▭	▭	●	▭
8. variant decrease	—	▭	▭	●	▭
9. variant decrease	—	▭	▭	●	▭
10. postcondition	—	▭	▭	●	▭
11. postcondition	—	▭	▭	●	▭

```
index_impl_po_1

t: global pointer
n: int
alloc: alloc_table
H1: valid_range(alloc, t, 0, n - 1)
________________

0 <= 0
```

```
/*@ requires \valid_range(t,0,n-1)
  @ ensures
  @   (0 <= \result < n => t[\result] == v) &&
  @   (\result == n => \forall int i; 0 <= i < n => t[i] != v)
  @*/
int index(int t[], int n, int v) {
  int i = 0;
  /*@ invariant 0 <= i && \forall int k; 0 <= k < i => t[k] != v
    @ variant n - i */
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

Timeout 10 ☐ Pretty Printer | file: ex1.c Correctness of C function index

Programming by Contract

- Contract:
  - Write a program P which computes a number *y* whose square is less than the input *x*
  - If the input *x* is a positive number, compute a number whose square is less than *x*
- Hoare Triples

Pre-condition Post-condition

  - $(|\phi|)P(|\psi|)$
  - Program P is run in a state that satisfies ϕ, then the sate after it executes will satisfy ψ
  - $(|x>0|)\ P(|y \bullet y<x|)$

Slides from CS453 Fall 2005 taught by Prof.Key-Sun Choi

Program Verification through Programming by Contract/Theorem Proving

- The earliest scientific approach to verify a target software
- Requires human expertise on the target software
  - If a user can specify important characteristic of the target SW in a "good" way, proof can succeed.
    - Ex. Loop invariant
  - Note that computer scientists in early days were mathematicians and logicians
- Not automatic, but the verification result is general (i.e. not bounded within n <= 10)

Proof rules for partial correctness of Hoare triples

$$\frac{(|\phi|)C_1(|\eta|)\;(|\eta|)C_2(|\psi|)}{(|\phi|)C_1;C_2(|\psi|)}\ \text{Composition}$$

$$\frac{}{(|\psi[E/x]|)x=E(|\psi|)}\text{Assignment}$$

$$\frac{(|\phi\wedge \mathrm{B}|)C_1(|\psi|)\;(|\phi\wedge\neg B|)C_2(|\psi|)}{(|\phi|)\mathrm{if}\,B\{C_1\}\mathrm{else}\{C_2\}(|\psi|)}\text{If - statement}$$

$$\frac{(|\psi\wedge B|)C(|\psi|)}{(|\psi|)\mathrm{while}B\{C\}(|\psi\wedge\neg B|)}\text{Partial - while}$$

$$\frac{\mapsto_{\mathrm{AR}}\phi\to\phi\;(|\phi|)C(|\psi|)\;\mapsto_{AR}\psi\to\psi}{(|\phi\ |)C(|\psi\ |)}\text{Implied}$$

Assignment

$$\boxed{\dfrac{}{(\!|\psi[E/x]|\!)\; x = E\; (\!|\psi|\!)}}$$

- $\psi[E/x]$
  - Denotes the formula obtained by taking ψ and replacing all free occurrences of x with E
  - ψ with E in place of x - whatever ψ says about x but applied to E - must be true in the initial state
- *Backward verification* for $(|\ \psi[E/x]\ |)\ x{=}E\ (|\ \psi\ |)$
  - If we know ψ and wish to find ϕ such that

    $(|\ \phi\ |)\ x{=}E\ (|\ \psi\ |)$

Examples

- If P: x=2, then are the followings true?
  a) $(|2=2|)P(|x=2|)$
  b) $(|2=4|)P(|x=4|)$
    - ✓ $(|\perp|)\ x=E\ (|\psi|)$
  c) $(|2=y|)P(|x=y|)$
  d) $(|2>0|)P(|x>0|)$

- P: x=x+1
  a) $(|x+1=2|)P(|x=2|)$
  b) $(|x+1=y|)P(|x=y|)$
  c) $(|x+1+5=y|)P(|x+5=y|)$
  d) $(|x+1>0 \wedge y>0|)P(|x>0 \wedge y>0|)$

If-statements

$$\frac{(\!|\phi \wedge \mathrm{B}|\!)\, C_1\, (\!|\psi|\!) \qquad (\!|\phi \wedge \neg B|\!)\, C_2\, (\!|\psi|\!)}{(\!|\phi|\!)\, \mathrm{if}\ B\, \{C_1\}\, \mathrm{else}\, \{C_2\}\, (\!|\psi|\!)}$$

- $(| \phi |)$ if B $\{C_1\}$ else $\{C_2\}(/ \psi |)$
  - Decompose it into two triples, subgoals corresponding to the cases of *B* = true and false

While-statements

$$\frac{(\!|\psi \wedge B|\!)\, C\, (\!|\psi|\!)}{(\!|\psi|\!)\, \text{while}\, B\, \{C\}\, (\!|\psi \wedge \neg B|\!)}$$

- Invariant ψ
- No matter how many times the body *C* is executed, if ψ is true initially and the while-statement terminates, then ψ will be true at the end.
- Since the while-statement has terminated, *B* will be false.

Implied

$$\frac{\vdash_{AR} \phi' \rightarrow \phi \quad (|\phi|)\, C\, (|\psi|) \quad \vdash_{AR} \psi \rightarrow \psi'}{(|\phi'|)\, C\, (|\psi'|)}$$

- A sequent $|\text{-}_{AR}\ \phi \rightarrow \phi'$ is valid iff there is a proof of ϕ' in the natural deduction calculus for predicate logic, where ϕ and standard laws of arithmetic are premises.
- Precondition – strengthened
  - In general, we want weakest pre-condition to make a proof as general as possible
- Postcondition – weakened
  - In general, we want strongest post-condition to make a proof as general as possible

Partial-correctness proof for Fac1 in tree form

- (|T|)Fac1(|y=x!|)

$$\dfrac{\dfrac{\dfrac{(\!|1=1|\!)\,y=1\,(\!|y=1|\!)}{(\!|\mathsf{T}|\!)\,y=1\,(\!|y=1|\!)}\,i \quad \dfrac{(\!|y=1\wedge 0=0|\!)\,z=0\,(\!|y=1\wedge z=0|\!)}{(\!|y=1|\!)\,z=0\,(\!|y=1\wedge z=0|\!)}\,i}{(\!|\mathsf{T}|\!)\,y=1;z=0\,(\!|y=1\wedge z=0|\!)}\,c \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\!|y\cdot(z+1)=(z+1)!|\!)\,z=z+1\,(\!|y\cdot z=z!|\!)}{(\!|y=z!\wedge z\neq x|\!)\,z=z+1\,(\!|y\cdot z=z!|\!)}\,i \quad \dfrac{}{(\!|y\cdot z=z!|\!)\,y=y*z\,(\!|y=z!|\!)}}{(\!|y=z!\wedge z\neq x|\!)\,z=z+1;y=y*z\,(\!|y=z!|\!)}\,c}{(\!|y=z!|\!)\,\mathit{while}(z!=x)\{z=z+1;y=y*z\}\,(\!|y=z!\wedge z=x|\!)}\,w}{(\!|y=1\wedge z=0|\!)\,\mathit{while}(z!=x)\{z=z+1;y=y*z\}\,(\!|y=x!|\!)}\,i}{(\!|\mathsf{T}|\!)\,y=1;z=0;\mathit{while}(z!=x)\{z=z+1;y=y*z\}\,(\!|y=x!|\!)}\,c$$

```
Program Fac1:
        y=1 ;
        z=0;
        while (z != x) {
                z=z+1;
                y=y*z;
        }
```

Proof Strategies

- How should the intermediate formulas ϕ_i be found?
  - Backward works for assignment rule
  - Weakest precondition of C_{i+1}, given the postcondition ϕ_{i+1}
- Proof is constructed bottom-up
  - Justification makes sense when read top-down
  - The weakest precondition ϕ' is then checked to see whether it follows from the given precondition ϕ.
  - We appeal to the **Implied** rule.
    - An interface between predicate logic with arithmetic and program logic

Examples 4.13.1

$\vdash_{par} (|y=5|)\ x=y+1\ (|x=6|)$

$(|y=5|)$

$(|y+1=6|)$ **Implied**

x=y+1

$(|x=6|)$ **Assignment**

- Proof is constructed from the Bottom upwards.

Example 4.13.3

- Goal is to show that *u* stores the sum of *x* and *y* after the following sequence of assignments terminates.

```
z = x;
z = z + y;
u = z;
```

- Proof backwards

$(|T|)$

$(|x+y=x+y|)$ Imlied

z = x;

$(|z+y=x+y|)$ Assignment

z = z+y;

$(|z =x+y|)$ Assignment

u = z;

$(|u=x+y|)$ Assignment

Example 4.14: *If-statements*

```
a = x +1;
if (a - 1 == 0) {
   y = 1;
} else {
   y =a;
}
```

ϕ_1 is $1 = x+1$

ϕ_2 is $a = x+1$

$$\frac{\llparenthesis \phi_1 \rrparenthesis\, C_1\, \llparenthesis \psi \rrparenthesis \quad \llparenthesis \phi_2 \rrparenthesis\, C_2\, \llparenthesis \psi \rrparenthesis}{\llparenthesis (B \to \phi_1) \wedge (\neg B \to \phi_2) \rrparenthesis\, \mathrm{if}\ B\, \{C_1\}\, \mathrm{else}\, \{C_2\}\, \llparenthesis \psi \rrparenthesis}$$

$$\frac{\llparenthesis \phi \wedge \mathrm{B} \rrparenthesis\, C_1\, \llparenthesis \psi \rrparenthesis \quad \llparenthesis \phi \wedge \neg B \rrparenthesis\, C_2\, \llparenthesis \psi \rrparenthesis}{\llparenthesis \phi \rrparenthesis\, \mathrm{if}\ B\, \{C_1\}\, \mathrm{else}\, \{C_2\}\, \llparenthesis \psi \rrparenthesis}$$

Partial-While

$$\boxed{\frac{(\!|\eta \wedge B|\!)\, C\, (\!|\eta|\!)}{(\!|\eta|\!)\, while\, B\{C\}\, (\!|\eta \wedge \neg B|\!)}}$$

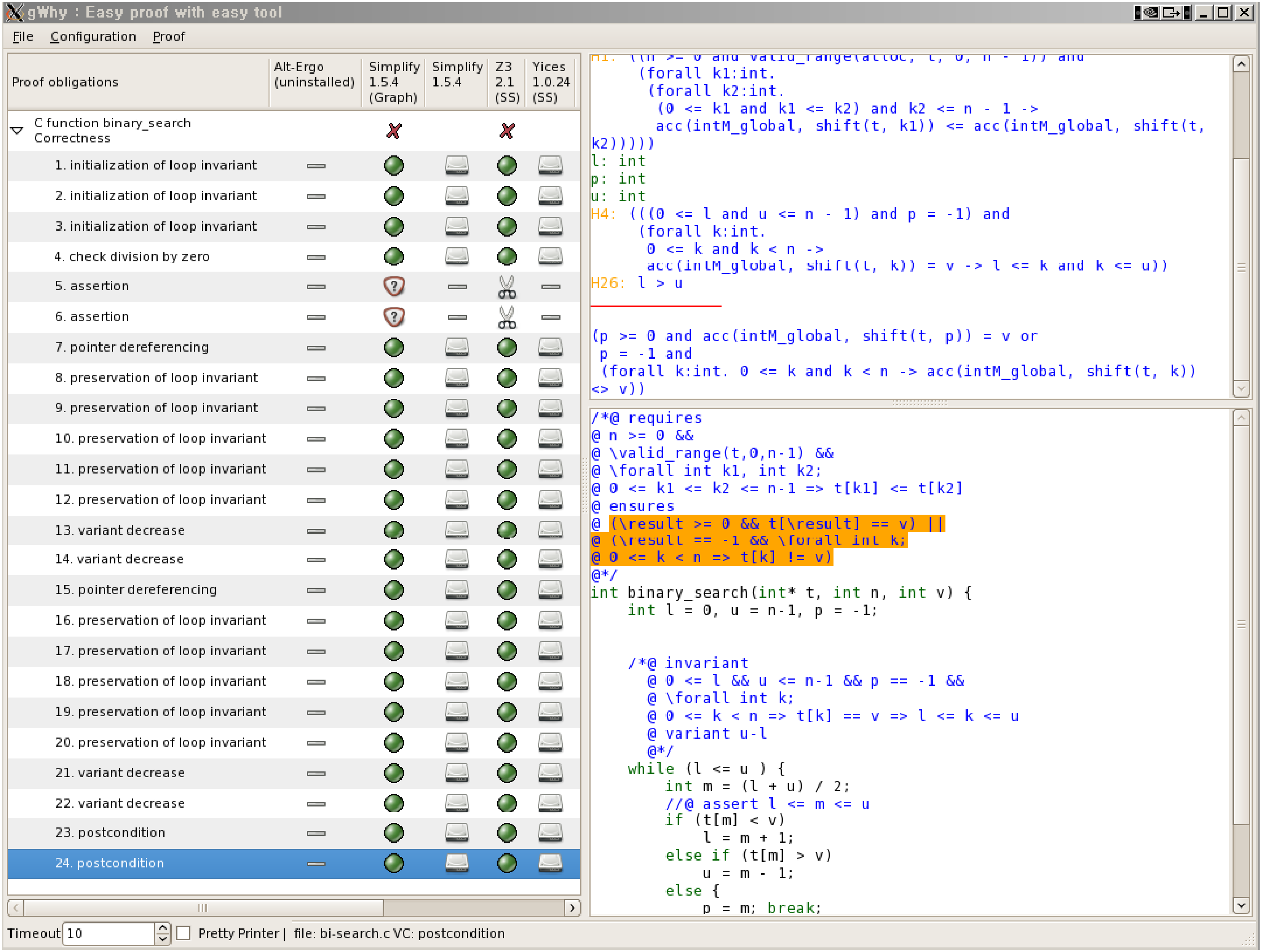
- η is invariant.
- (|φ|) while (B) {C} (|ψ|)
  - φ and ψ are not related.
  - How to relate? -- Discover a suitable η, such that
    - $\vdash \phi \rightarrow \eta$
    - $\vdash \eta \wedge \neg B \rightarrow \psi$
    - (| η |) while (B) {C} (| η ∧¬B |) hold.
  - “Implied-rule” discovery
    - Dijkstra

Binary Search Example

```
// Note that requires/ensures can access only function parameters and return value
/*@ requires
@ n >= 0 && \valid_range(t,0,n-1) &&
@ \forall int k1, int k2;
@ 0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
@ ensures
@ (\result >= 0 && t[\result] == v) ||
@ (\result == -1 &&
@    \forall int k;  0 <= k < n => t[k] != v)
@*/
```

```
int binary_search(int* t, int n, int v) {
   int l = 0, u = n-1, p = -1;
   /*@ invariant
     @ 0 <= l && u <= n-1 && p == -1 &&
     @ \forall int k;
     @ 0 <= k < n => t[k] == v => l <= k <= u
     @ variant u-l
     @*/
   while (l <= u ) {
      int m = (l + u) / 2;
      //@ assert l <= m <= u
      if (t[m] < v)
         l = m + 1;
      else if (t[m] > v)
         u = m - 1;
      else {
         p = m; break;
      }
   }
   return p;
}
```

Snapshot of WHY Result



Proof obligations	Alt-Ergo (uninstalled)	Simplify 1.5.4 (Graph)	Simplify 1.5.4	Z3 2.1 (SS)	Yices 1.0.24 (SS)
C function binary_search Correctness		✗		✗	
1. initialization of loop invariant	—	●	▭	●	▭
2. initialization of loop invariant	—	●	▭	●	▭
3. initialization of loop invariant	—	●	▭	●	▭
4. check division by zero	—	●	▭	●	▭
5. assertion	—	?	—	✂	—
6. assertion	—	?	—	✂	—
7. pointer dereferencing	—	●	▭	●	▭
8. preservation of loop invariant	—	●	▭	●	▭
9. preservation of loop invariant	—	●	▭	●	▭
10. preservation of loop invariant	—	●	▭	●	▭
11. preservation of loop invariant	—	●	▭	●	▭
12. preservation of loop invariant	—	●	▭	●	▭
13. variant decrease	—	●	▭	●	▭
14. variant decrease	—	●	▭	●	▭
15. pointer dereferencing	—	●	▭	●	▭
16. preservation of loop invariant	—	●	▭	●	▭
17. preservation of loop invariant	—	●	▭	●	▭
18. preservation of loop invariant	—	●	▭	●	▭
19. preservation of loop invariant	—	●	▭	●	▭
20. preservation of loop invariant	—	●	▭	●	▭
21. variant decrease	—	●	▭	●	▭
22. variant decrease	—	●	▭	●	▭
23. postcondition	—	●	▭	●	▭
24. postcondition	—	●	▭	●	▭

```
H1: ((n >= 0 and valid_range(alloc, t, 0, n - 1)) and
     (forall k1:int.
      (forall k2:int.
       (0 <= k1 and k1 <= k2) and k2 <= n - 1 ->
       acc(intM_global, shift(t, k1)) <= acc(intM_global, shift(t,
k2)))))
l: int
p: int
u: int
H4: (((0 <= l and u <= n - 1) and p = -1) and
     (forall k:int.
      0 <= k and k < n ->
      acc(intM_global, shift(t, k)) = v -> l <= k and k <= u))
H26: l > u


(p >= 0 and acc(intM_global, shift(t, p)) = v or
 p = -1 and
 (forall k:int. 0 <= k and k < n -> acc(intM_global, shift(t, k))
<> v))
```

```
/*@ requires
@ n >= 0 &&
@ \valid_range(t,0,n-1) &&
@ \forall int k1, int k2;
@ 0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
@ ensures
@ (\result >= 0 && t[\result] == v) ||
@ (\result == -1 && \forall int k;
@ 0 <= k < n => t[k] != v)
@*/
int binary_search(int* t, int n, int v) {
    int l = 0, u = n-1, p = -1;


    /*@ invariant
      @ 0 <= l && u <= n-1 && p == -1 &&
      @ \forall int k;
      @ 0 <= k < n => t[k] == v => l <= k <= u
      @ variant u-l
      @*/
    while (l <= u ) {
        int m = (l + u) / 2;
        //@ assert l <= m <= u
        if (t[m] < v)
            l = m + 1;
        else if (t[m] > v)
            u = m - 1;
        else {
            p = m; break;
```

Timeout 10 ☐ Pretty Printer | file: bi-search.c VC: postcondition