Queens on a Chessboard

an exercise in program verification

Jean-Christophe Filliâtre

Krakatoa/Caduceus working group

December 15th, 2006

Introduction

challenge for **the verified program of the month**:

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

appears on a web page collecting C signature programs

due to Marcel van Kervinck,
author of MSCP (Marcel's Simple Chess Program)

Introduction

challenge for **the verified program of the month**:

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

appears on a web page collecting C signature programs

due to Marcel van Kervinck,
author of MSCP (Marcel's Simple Chess Program)

Unobfuscating...

```
int t(int a, int b, int c) {
  int d=0, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=(e-=d)&-e;)
      f+=t(a-d,(b+d)*2,(c+d)/2);
  return f;
}

int main(int q) {
  scanf("%d",&q);
  printf("%d\n",t(~(~0<<q),0,0));
}
```

Unobfuscating...

```
int t(int a, int b, int c) {
  int d=0, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=(e-=d)&-e;)
      f+=t(a-d,(b+d)*2,(c+d)/2);
  return f;
}

int f(int n) {
  return t(~(~0<<n), 0, 0);
}
```

we end up with a mysterious function f : $\mathbb{N} \rightarrow \mathbb{N}$

Queens on a chessboard

given a number n smaller than 32, `f`(n) is the number of ways to put n queens on $n \times n$ chessboard so that they cannot beat each other

let us prove that this program is **correct**, that is:

- it does not crash
- it terminates
- it computes the right number

Queens on a chessboard

given a number n smaller than 32, `f`(n) is the number of ways to put n queens on $n \times n$ chessboard so that they cannot beat each other

let us prove that this program is **correct**, that is:

- it does not crash
- it terminates
- it computes the right number

Why is it challenging?

in two lines of code we have

- C idiomatic bitwise operations
- loops & recursion, involved in a backtracking algorithm
- highly efficient code

How does it work?

- backtracking algorithm (no better way to solve the N queens)
- integers used as **sets** (bit vectors)

integers	sets
`0`	$\emptyset$
`a&b`	$a \cap b$
`a+b`	$a \cup b$, when $a \cap b = \emptyset$
`a-b`	$a \setminus b$, when $b \subseteq a$
`~a`	$\complement a$
`a&-a`	$min\_elt(a)$, when $a \neq \emptyset$
`~(~0<<n)`	$\{0, 1, \ldots, n-1\}$
`a*2`	$\{i+1 \mid i \in a\}$, written $S(a)$
`a/2`	$\{i-1 \mid i \in a \wedge i \neq 0\}$, written $P(a)$

How does it work?

- backtracking algorithm (no better way to solve the N queens)
- integers used as **sets** (bit vectors)

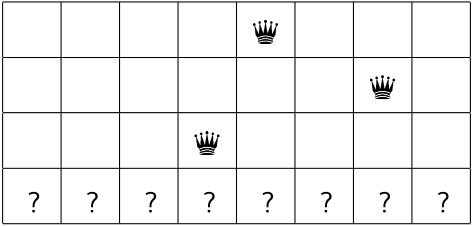
integers	sets
`0`	$\emptyset$
`a&b`	$a \cap b$
`a+b`	$a \cup b$, when $a \cap b = \emptyset$
`a-b`	$a \setminus b$, when $b \subseteq a$
`~a`	$\complement a$
`a&-a`	$min\_elt(a)$, when $a \neq \emptyset$
`~(~0<<n)`	$\{0, 1, \ldots, n-1\}$
`a*2`	$\{i+1 \mid i \in a\}$, written $S(a)$
`a/2`	$\{i-1 \mid i \in a \land i \neq 0\}$, written $P(a)$

Code rephrased on sets

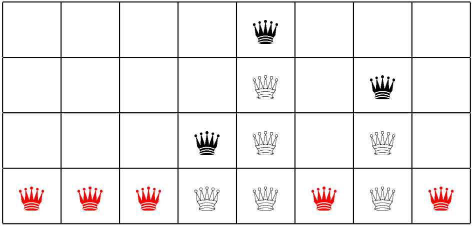
```
int t(a, b, c)
  if a ≠ ∅
    e ← (a\b)\c
    f ← 0
    while e ≠ ∅
      d ← min_elt(e)
      f ← f + t(a\{d}, S(b ∪ {d}), P(c ∪ {d}))
      e ← e\{d}
    return f
  else
    return 1

int f(n)
  return t({0,1,...,n−1}, ∅, ∅)
```

What a, b and c mean

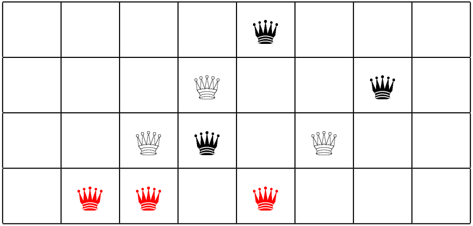


What a, b and c mean



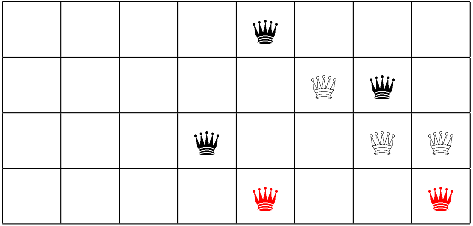
a = columns to be filled = 11100101_2

What a, b and c mean



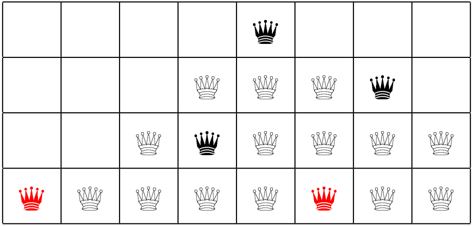
b = positions to avoid because of left diagonals $= 01101000_2$

What a, b and c mean



c = positions to avoid because of right diagonals = 00001001_2

What a, b and c mean



$a \& \tilde{} b \& \tilde{} c$ = positions to try = 10000100_2

Now it is clear

```
int t(int a, int b, int c) {
  int d=0, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=(e-=d)&-e;)
      f += t(a-d,(b+d)*2,(c+d)/2);
  return f;
}

int f(int n) {
  return t(~(~0<<n), 0, 0);
}
```

Abstract finite sets

```
//@ type iset
```

```
//@ predicate in_(int x, iset s)
```

```
/*@ predicate included(iset a, iset b)
  @  { \forall int i; in_(i,a) => in_(i,b) } */
```

```
//@ logic iset empty()
```

```
//@ axiom empty_def : \forall int i; !in_(i,empty())
```

...

total: **66 lines** of functions, predicates and axioms

C ints as abstract sets

```
//@ logic iset iset(int x)
```

```
/*@ axiom iset_c_zero : \forall int x;
  @   iset(x)==empty() ⇔ x==0 */
```

```
/*@ axiom iset_c_min_elt :
  @   \forall int x; x != 0 ⇒
  @     iset(x&-x) == singleton(min_elt(iset(x))) */
```

```
/*@ axiom iset_c_diff : \forall int a, int b;
  @   iset(a&~b) == diff(iset(a), iset(b)) */
```

. . .

total: **27 lines** / should be proved independently

Warmup: termination of the for loop

```
int t(int a, int b, int c){
  int d=0, e=a&~b&~c, f=1;
  if (a)
    //@ variant card(iset(e-d))
    for (f=0; d=(e-=d)&-e; ) {
      f += t(a-d,(b+d)*2,(c+d)/2);
    }
  return f;
}
```

3 verification conditions, all proved automatically

Warmup: termination of the recursive function

```
int t(int a, int b, int c){
  int d=0, e=a&~b&~c, f=1;
  //@ label L
  if (a)
    /*@ invariant
      @   included(iset(e-d), iset(e)) &&
      @   included(iset(e),\at(iset(e),L))
      @*/
    for (f=0; d=(e-=d)&-e; ) {
      /*@ assert \exists int x;
        @   iset(d) == singleton(x) && in_(x,iset(e)) */
      //@ assert card(iset(a-d)) < card(iset(a))
      f += t(a-d,(b+d)*2,(c+d)/2);
    }
  return f;
}
```

7 verification conditions, all proved automatically

Soundness

how to express that we compute the right number,
since the program is not storing anything,
not even the current solution?

answer: by introducing **ghost code** to perform the missing operations

Soundness

how to express that we compute the right number,
since the program is not storing anything,
not even the current solution?

answer: by introducing **ghost code** to perform the missing operations

Ghost code

ghost code can be regarded as regular code, as soon as

- ghost code does not modify program data
- program code does not access ghost data

ghost data is purely logical $\Rightarrow$ ne need to check the validity of pointers

ghost code is currently restricted in Caduceus, but should not be

Program instrumented with ghost code

```
//@ int** sol;
//@ int s;
//@ int* col;
//@ int k;

int t(int a, int b, int c){
  int d=0, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=(e-=d)&-e; ) {
      //@ col[k] = min_elt(d);
      //@ k++;
      f += t3(a-d, (b+d)*2, (c+d)/2);
      //@ k--;
    }
  //@ else
  //@   store_solution();
  return f;
}
```

Annotations (1/4)

```
/*@ requires solution(col)
  @ assigns  s, sol[s][0..N()-1]
  @ ensures  s==\old(s)+1 && eq_sol(sol[\old(s)], col)
  @*/
void store_solution();

/*@ requires
  @   n == N() && s == 0 && k == 0
  @ ensures
  @   \result == s &&
  @   \forall int* t; solution(t) ⇔
  @       (\exists int i; 0≤i<\result && eq_sol(t,sol[i]))
  @*/
int queens(int n) {
  return t(~(~0<<n),0,0);
}
```

Annotations (2/4)

```
//@ logic int N()
```

```
/*@ predicate partial_solution(int k, int* s) {
  @   \forall int i; 0 ≤ i < k ⇒
  @     0 ≤ s[i] < N() &&
  @     (\forall int j; 0 ≤ j < i ⇒ s[i] != s[j] &&
  @                                 s[i]-s[j] != i-j &&
  @                                 s[i]-s[j] != j-i)
  @ }
  @*/
```

```
//@ predicate solution(int* s) { partial_solution(N(), s) }
```

Annotations (3/4): specification of t

```
/*@ requires
  @   0 ≤ k && k + card(iset(a)) == N() && 0 ≤ s &&
  @   pre_a:: (\forall int i; in_(i,iset(a)) ⇔
  @             (0≤i<N() && \forall int j; 0≤j<k ⇒ i != col[j]))
  @   pre_b:: (\forall int i; i>=0 ⇒ (in_(i,iset(b)) ⇔
  @             (\exists int j; 0≤j<k && col[j] == i+j-k))) &&
  @   pre_c:: (\forall int i; i>=0 ⇒ (in_(i,iset(c)) ⇔
  @             (\exists int j; 0≤j<k && col[j] == i+k-j))) &&
  @   partial_solution(k, col)
  @ assigns
  @   col[k..], s, k, sol[s..][..]
  @ ensures
  @   \result == s - \old(s) && \result >= 0 && k == \old(k) &&
  @   \forall int* t;
  @      ((solution(t) && eq_prefix(col,t,k)) ⇔
  @      (\exists int i; \old(s)≤i<s && eq_sol(t, sol[i])))
  @*/
```

Annotations (4/4): loop invariant

```
/*@ invariant
  @   included(iset(e-d),iset(e)) &&
  @   included(iset(e),\at(iset(e),L)) &&
  @   f == s - \at(s,L) && f >= 0 && k == \old(k) &&
  @   partial_solution(k, col) &&
  @   \forall int *t;
  @     (solution(t) &&
  @      \exists int di; in_(di, diff(iset(e),\at(iset(e),L))) &&
  @        eq_prefix(col,t,k) && t[k]==di) ⇔
  @     (\exists int i; \at(s,L)≤i<s && eq_sol(t, sol[i]))
  @ loop_assigns
  @   col[k..], s, k, sol[s..][..]
  @*/
for (f=0; d=(e-=d)&-e; ) {
  ...
```

Finally, we get...

256 lines of code and specification

on a slightly more abstract Why version of the program:

- main function `queens`: **15** verification conditions
  - **all** proved automatically (Simplify, Ergo or Yices)
- recursive function `t`: **51** verification conditions
  - **42** proved automatically: 41 by Simplify, 37 by Ergo and 35 by Yices
  - **9** proved manually using Coq (and Simplify)

Finally, we get...

256 lines of code and specification

on a slightly more abstract Why version of the program:

- main function `queens`: **15** verification conditions
  - **all** proved automatically (Simplify, Ergo or Yices)
- recursive function `t`: **51** verification conditions
  - **42** proved automatically: 41 by Simplify, 37 by Ergo and 35 by Yices
  - **9** proved manually using Coq (and Simplify)

Missing

we need to show that there is no duplicate among the solutions

improve the results on the C version:

- `queens`: 13/15
- `t`: 39/54 (72% instead of 80%)

Missing

we need to show that there is no duplicate among the solutions

improve the results on the C version:

- `queens`: 13/15
- `t`: 39/54 (72% instead of 80%)

Overflows

the requirement $n < 32$ is not an issue
world record is $n = 25$
all computers will be 64 bits before we reach $n = 32$

but the program contains unrelevant overflows when $n \geq 16$
thus ensuring the absence of overflows would require $n < 16$
we need a **model of overflows** for this program

Overflows

the requirement $n < 32$ is not an issue
world record is $n = 25$
all computers will be 64 bits before we reach $n = 32$

but the program contains unrelevant overflows when $n \geq 16$
thus ensuring the absence of overflows would require $n < 16$
we need a **model of overflows** for this program