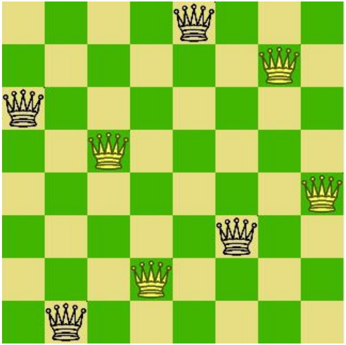
Guaranteed Primitive Redex Speculation (Work in Progress)

Colin Runciman

Department of Computer Science, University of York

Running Example: N-Queens Program



[6, 1, 5, 2, 8, 3, 7, 4]

Compute all solutions for a given no. of queens:

`queens :: Int -> [[Int]]`

Primitive Speculation Illustrated

Consider the `safe` function:

```
safe :: Int -> Int -> [Int] -> Bool
safe x d []       =  True
safe x d (q:qs)   =  x /= q && x /= q+d && x /= q-d &&
                     safe x (d+1) qs
```

Tracing the program by hand, we need to evaluate `safe 1 1 [2]`. Do we use mere substitution

```
1 /= 2 && 1 /= 2+1 && 1 /= 2-1 && safe 1 (1+1) []
```

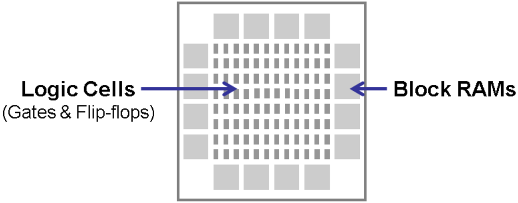
or a little speculation

```
True && 1 /= 3 && 1 /= 1 && safe 1 2 []
```

and what does our computer do?

Context: The Reduceron

- The Reduceron is a graph-reduction machine, described by a functional program, and implemented using reconfigurable hardware (FPGA).



- The Reduceron works by template instantiation, reducing function applications by substituting arguments in bodies.

Reduceron Characteristics

- The size of compiled bodies is bounded so that by using wide parallel memories bodies are instantiated in a single clock cycle.

- Primitive redexes in instances of function bodies are detected dynamically for primitive redex speculation.

Reduceron Characteristics

- The size of compiled bodies is bounded so that by using wide parallel memories bodies are instantiated in a single clock cycle.
- Primitive redexes in instances of function bodies are detected dynamically for primitive redex speculation.
- The sizes of bodies containing primitive applications are reckoned as if every primitive-redex test fails.

Detecting Guaranteed PRS Candidates Statically

Goal

Find the primitive applications whose every run-time instance is guaranteed to be a redex.

Method?

Suppose we propagate integer-value information:

- inwards from program input;
- outwards from numeric literals;
- onwards through primitive redex speculation.

Example

In `safe` we find just one guaranteed primitive redex

```
safe x d (q:qs)   =   x /= q && x /= q+d && x /= q-d &&
                      safe x (d+1) qs
```

as both `x` and `q` are drawn from data structures.

Valuable data structures

Definition

Let D be a data expression that evaluates to the construction $C\ e_1 \ldots e_n$. D is valuable if each integer component e_i is a value, and each data component e_i is valuable.

Example

```
toOne :: Int -> [Int]
toOne n  =  if n==1 then [1] else n : toOne (n-1)
```

If `n` is a value, then `toOne n` is valuable.

Revisiting `safe`

With information about valuable data structures, the guaranteed primitive redexes become:

```
safe x d (q:qs)  =  x /= q && x /= q+d && x /= q-d &&
                    safe x (d+1) qs
```

Values in Higher-order Programs

The result just noted is for a first order version of queens.

A solution by comprehension

```
queens nq   =   gen nq nq

gen 0 nq    =   [[]]
gen n nq    =   [q:b | b <- gen (n-1) nq, q <- [1..nq],
                       safe q 1 b]
```

translates to applications of higher-order functions.

Valuable functions

Definition

A value function is a primitive. A valuable function gives a valuable result if each of its arguments is a value or valuable.

- Constructors are valuable.
- Partial applications of valuable functions to values and valuable arguments are valuable.

Example

```
foldr f z []        =  z
foldr f z (x:xs)    =  f x (foldr f z xs)

append xs ys        =  foldr (:) ys xs
concat              =  foldr append []
```

Can you verify that append and concat are both valuable?

Non-uniform Valuations — a Problem?

- What if for some applications of a function there is scope for primitive-redex speculation in the body but for others there is not? Or if in some cases a body is valuable, but in others not?
- No uniform guarantee can be given, but we don't want to lose speculative evaluation in the cases where it is possible.

Example

```
toOne :: Int -> [Int]
toOne n  =  if n==1 then [1] else n : toOne (n-1)
```

- In one place `toOne 8`.
- In another `toOne (length (queens 8))`.

Cloning and specialization

- Solution: clone by need, specialising functions for different combinations of value/valuable argument positions.
- In principle, the number of clones could be exponential in the arity of a function. In practice, there is often just one specialization needed — and the original is discarded.

Recall:

```
toOne :: Int -> [Int]
toOne n  =  if n==1 then [1] else n : toOne (n-1)
```

- Original: `n` might not be a value; the result is not valuable; the function is recursive with argument `n-1` passed unevaluated.
- Clone: `n` is a value; the result is valuable; the function is recursive with argument `n-1` reduced speculatively.

Value and Strictness

- An n-ary function f is strict in its m^{th} argument if $f\ e_1 \ldots e_{m-1} \perp e_{m+1} \ldots e_n = \perp$.
- Since the early '80s optimizing functional-language compilers have used strictness to justify eager evaluation, avoiding the work of building expressions on the heap.
- Analysis of deeper forms of strictness for data structures and functions is notoriously expensive, and usually not attempted.

Applicability in N-Queens

The `safe` function

```
safe x d (q:qs)  =  x /= q && x /= q+d && x /= q-d &&
                    safe x (d+1) qs
```

is strict in `x` — but `x` is invariant. It is not strict in `d`. Nor is it spine-strict in the list argument. Not much help!

Value and Type

- Since the early '90s, some lazy functional languages or compilers allow distinct types for unboxed values such as integers never stored as unevaluated expressions.
- The worker-wrapper transformation★ can introduce unboxed types automatically.

Applicability in N-Queens

A worker for the `safe` function might be

```
safe :: #Int -> #Int -> [Int] -> Bool
safe x d (q:qs)  =  let q' = value q in
                    x /= q' && x /= q'+d && x /= q'-d &&
                    safe x (d+1) qs
```

but unboxing of `q` values is likely to require explicit programming.

★ Courtesy reference to Graham H's work!

Performance Results

The current dynamic implementation of primitive-redex speculation gives a 2$\times$ speedup for `queens`.

There is only a prototype of the first-order value analysis, with specialisation of clones. Higher-order analysis and the adaptation of the Reduceron for guaranteed primitive redexes are yet to be implemented.

Performance Results

The current dynamic implementation of primitive-redex speculation gives a 2× speedup for `queens`.

There is only a prototype of the first-order value analysis, with specialisation of clones. Higher-order analysis and the adaptation of the Reduceron for guaranteed primitive redexes are yet to be implemented.

But I expect another 2× speedup for `queens`!

Acknowledgements

The Reduceron project is joint work,
with post-doc researcher
Matthew Naylor
as the principal architect and builder,
and funding from

EPSRC

Engineering and Physical Sciences
Research Council