Termination checking for a lazy functional language

Neil Mitchell

Overview

- Background
  - Properties of functional languages
  - Bottom ⊥, Lazy, Higher order...
- Total programming
- Sized Types
- Termination Checkers
- Open questions

Bottom ⊥

```
head [1,2,3] = 1
head [] = ⊥
```

Not case complete – unspecified in some situations

```
sum [1..10] = 55
sum [1..] = ⊥
```

Never terminates, no error returned

Laziness

What is the result of head `[1..]`?

Strict: ⊥

Eager languages, C, ML, Scheme

Lazy: `1`

Haskell, Clean

`take 10 primes`

Higher Order

- Can pass a function as a value
- Possible to define a function `apply` such that:

```
sum     = apply add
product = apply multiply

apply f [x] = x
apply f (x:xs) = f x (apply f xs)
```

Total Functional Programming

Turner 1995, 2004 - of SASL, KRC, Miranda

- Functional programming without ⊥
  - Can't crash (case complete)
  - Can't loop forever (...unproductively)

- Requires *syntactic descent*

```
fact 0       = 1
fact (x+1) = (x+1) * fact x
```

λ Infinite and Total?

Telford and Turner 1997, 2000

- Useful for
  - Infinite lists – the list of primes
  - Reactive systems – embedded systems
  - Stream processing
- Use **codata** instead of data
- Keep codata and data separate
- Must be **productive**
  - Must generate next element in finite time
  - But can continually generate next elem

λ The Downside

- But total functional programming is not all good...
- **Not** Turing Complete
- Requires substantial rewrites to code
- Natural definitions are not correct
  - Need `map` and `comap`
  - Can't have `head`

```
first_even = head evens
```

Head v2.0

```
head []     = error "No head!"
head (x:xs) = x

head []     = Nothing
head (x:xs) = Just x

head a []     = a
head a (x:xs) = x

head no yes []     = no
head no yes (x:xs) = yes x
```

Sized Types

Hughes et al. 1996; Pareto 1998; Abel 2003

- Annotate type signatures with **size**
- Numbers become lists
  - Use `succ(x)` and `zero` – Peano numbers
  - 4 = succ(succ(succ(succ(zero))))

```
append :: [x] -> [x] ->  [x]
append ::  a  ->  b  -> a + b
```

Used to prove termination and productivity, composes upwards

Sized Types - Sorting

```
isort [] = []
isort (x:xs) = insert x (isort xs)

insert n [] = [n]
insert n (x:xs) = if n<=x
         then n: x: xs
         else x: insert n xs

isort  :: n -> n
insert :: _ -> n -> n + 1
```

Sized Types – Sorting (2)

```
qsort []     = []
qsort (x:xs) = qsort l++[x]++qsort h
    where l = filter (<= x) xs
          h = filter (> x) xs
```

```
filter :: _ -> n -> n
```

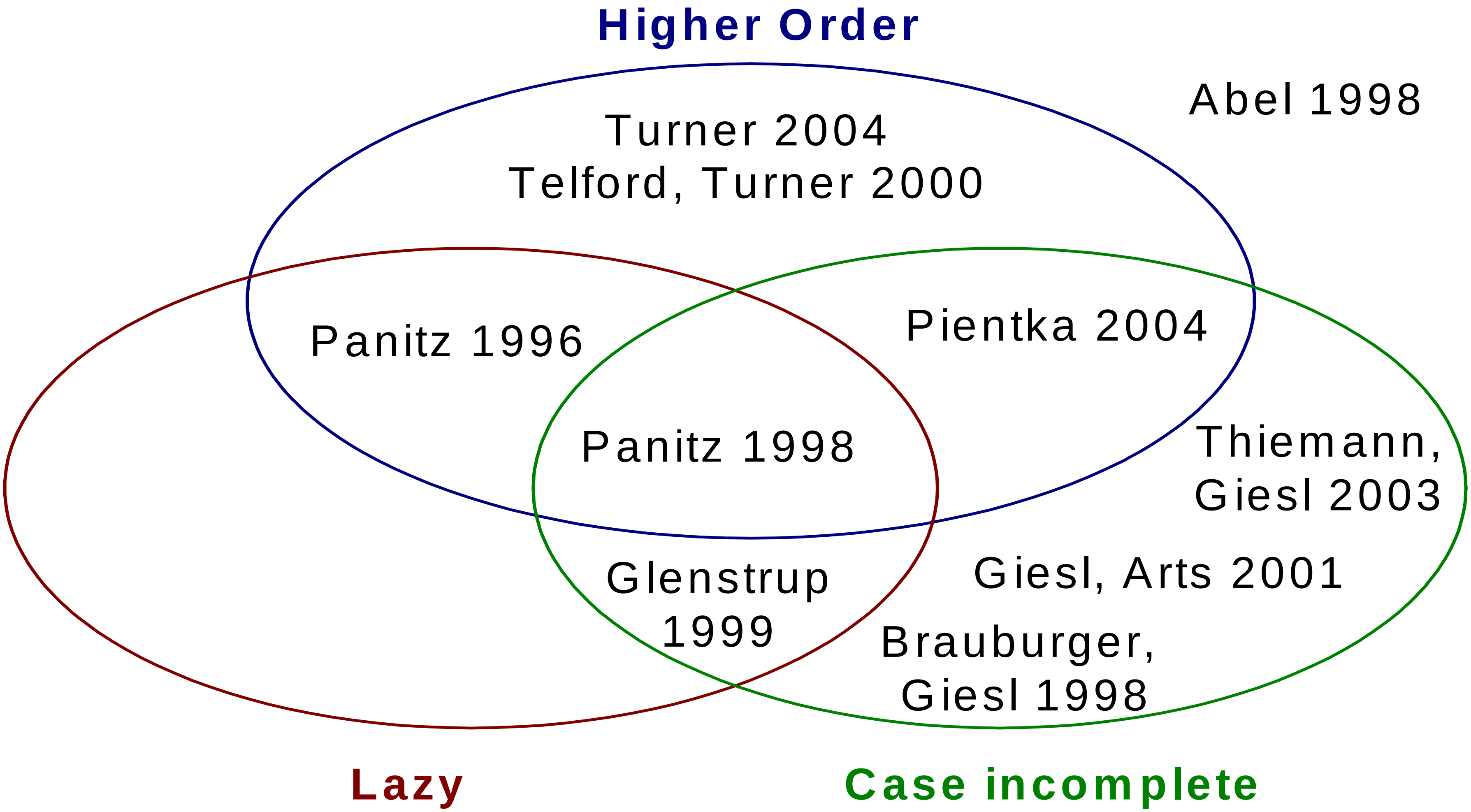
or $\leq$ n

```
qsort :: n -> ?
l / h :: n – 1
```

`qsort :: n ->` n^2

```
qsort :: n -> w
```

λ Termination Checkers



Higher Order

Abel 1998

Turner 2004
Telford, Turner 2000

Panitz 1996

Pientka 2004

Panitz 1998

Thiemann,
Giesl 2003

Glenstrup
1999

Giesl, Arts 2001

Brauburger,
Giesl 1998

Lazy

Case incomplete

Prolog termination checkers

Genaim, Codish 2001; Apt, Pedreschi 1993; Lindenstrauss, Sagiv 1997; Verbaeten et al 1991; lots more

- Properties of Prolog...
- Definitely case incomplete
- Higher order (using `call`) *Naish 96*
- Lazy?
  - Backtracking has similarities
  - Can encode laziness in Prolog

    Antoy, Hanus 2000

Prolog termination checkers (2)

- Lots of different methods
  - Most rely on building up an ordering over some term
  - Some use constraint solvers
  - Tabling, time complexity...
- There is a set of standard problems from various papers
  - Parsing, Ackermann, sort, reverse, greatest common divisor etc.
  - No solver gets them all!

Panitz 1998: TEA

- Translate to a Core language
- Use Tableau proof
  - Like case analysis
  - Variable *a* is either Nil, or Cons
- Looks for orderings on variables
- Errors as 'successful termination'
- '90%' successful

Normal Form (nf) Termination

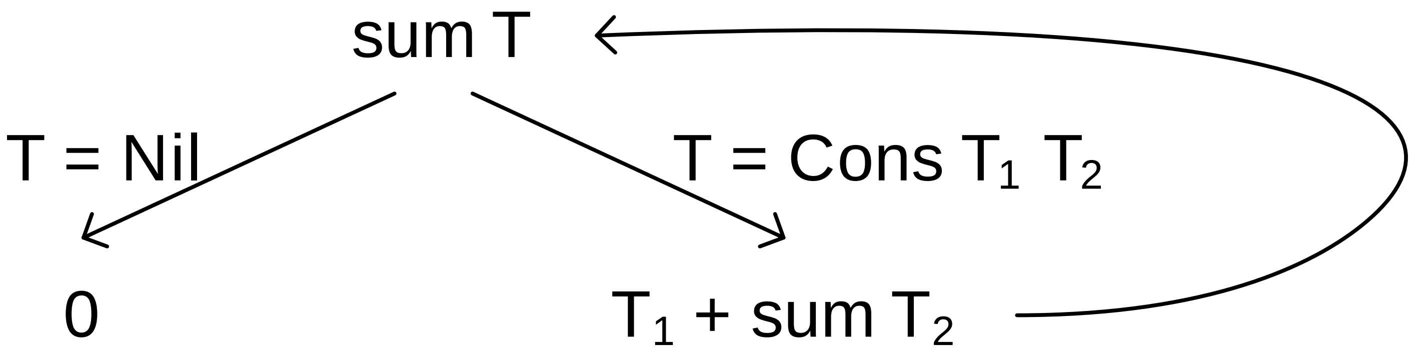
- An expression is in normal form if it cannot be reduced any further
  - `[1..]` does not have a normal form

- `f a b c` is nf-terminating if
  - Given a, b and c are in normal form
  - `f a b c` will reduce to normal form

- Proves nothing about `head [1..]`

Example: sum

```
sum Nil = 0
sum (Cons x xs) = x + sum xs
```



sum T

T = Nil

T = Cons T_1 T_2

0

T_1 + sum T_2

$T > T_2$

Cons T_1 T_2 $> T_2$

Cons a b = 1 + b

Open Questions

- Would a termination checker be used?
  - Maybe as part of a compiler?
  - Maybe for high quality code?

- How much code rewrite is acceptable?
  - None?
  - Just restricted to library functions?

Summary

- Properties of functional languages
  - Bottom ⊥, Lazy, Higher order
- Total programming
  - No ⊥, codata
- Sized Types
  - Extension of type system with size
- Termination Checkers
  - Prolog checkers
  - TEA: Haskell checker