A Dependently Typed Framework for Maintaining Invariants

László Németh

Department of Computer Science
Bilgi University, Istanbul

TYPES'06 - Nottingham

Introduction

Maintaining accurate information about programs is important

- Embedded systems: run-time and space behaviour, guarantees (size)
- Agressively optimising compilers: any predicate which allows transformation according to some criteria (speed, space)
- certified compilers: ’ rewrote application of `head` in line 42, because the list can never be empty’
- ultimately: certified code

Where to find this information?

- given some unannotated code, try to discover it: static analysis (difficult)
- force the user to annotate (or write it in a sufficiently rich language)
- ?

Haskell: Prelude vs Report

Legacy code: Haskell Prelude $\iff$ useful information: Report

Examples:

- 'The sequence `enumFromTo` $e_1 e_3$ is the list $[e_1, e_1 + 1, e_1 + 2, \ldots, e_3]$'. The list is empty if $e_1 > e_3$.

```
     ( R : A -> A -> * ;  xs : List A !
data !-------------------------------!
     !        Sorted R xs : *         )

                                  ( a : A ;  ss : Sorted R bs ;  p : Sm R a bs !
where (---------------------! ;  !-----------------------------------------!
      ! snil : Sorted R nil )   !    scons a ss p : Sorted R (cons a bs)     )
```

- 'The `Ix` class is used to map a contiguous subrange of values in a type onto integers. ... the nullary constructors are assumed to be numbered left-to-right with the indices being 0 to $n - 1$ inclusive'. : they are N's and SORTED
- careful reading of the Prelude+Report shows up dozens

Approach

1. Devise *enriched types which allows statically track invariants we are interested in*
2. *Transform well-typed Haskell functions to functions in a DT language which manipulates those invariants*
3. *Use the stronger type system to maintain and possibly establish properties*
4. *If typechecks extract*
   - *Haskell code + rewrite rules*
   - *Transformed code (beware of code duplication)*

In this talk

Two properties

- Size of lists
- (Sortedness)

We

- pretend that Haskell is strict
- deal only with a handful of functions from the Prelude ((++), filter, take, dropWhile, intersect, etc)
- use Epigram because it is theorem proving well disguised
- compiler = GHC

All (well-typed) combinations of those functions maintain the invariants

Size (dependently)

```
      ( A : * ;  lp : Le lb l ;  up : Le l ub !
data !-------------------------------------!
     !           BList A lp up : *          )


where (-------------------------! ;
       ! bnil : BList A leZ leZ )

       (      a : A ;  as : BList A lp up      !
       !---------------------------------------!
       ! bcons a as : BList A (leS lp) (leS up) )
```

Map Works

```
     ( f : all a : A => B ;  xs : BList A lp up !
let  !-----------------------------------------!
     !         bmap f xs : BList B lp up        )
bmap f xs <= rec xs
{ bmap f xs <= case xs
  { bmap f bnil => bnil
    bmap f (bcons x xs) => bcons (f x) (bmap f xs)
  }
}
```

and we get the usual goodies

```
     ( xs : BList A (leS n) u !
let  !-----------------------!
     !      bhead xs : A      )
bhead xs <= case xs
  { bhead (bcons m'' ms) => m''
```

Other Prelude Functions

- append adds the lower bounds and the upper bounds (think of the definition list comprehensions!)
- filter changes the proof about the lower bound to zero (ie the proof of)
- intersect sets the lower bound to zero and the upper bound to the minimum of the length of the lists
- take *n* sets the length to be exactly *n* if the lower bound is greater than *n*.
- . . .

What Transformations are Possible

```
take 3 ==>

lambda xs.
case xs of
  (x1:xs1) ->
    x1 : case xs1 of
           (x2:xs2) ->
             x2 : case xs2 of
                    (x3:xs3) -> x3 : []
```

Message

Instead of a whispering important properties (in the Report)

SAY

them in the types of (dependently typed) functions and make some compiler

SEE

them.

Summary

- more you say more you (should) get
- can be done under the hood (no extension to Core or Haskell itself)
- the approach complements the optimising capabilities of GHC

TODO

- PiE - formalise the Prelude (plus the Report!) in Epigram: good for the soul, good for compilation, and Haskell programmers can also read it
- once we established some invariant (and possibly acted upon) it is a shame to throw it away