Exploring the Type Inference Approach to Deforestation

Marie-Christine (Kirsten) Chevalier

Submitted in Partial Fulfillment of the Prerequisite for
Honors in Computer Science

May 2001

© 2001 Marie-Christine Chevalier

The tree which moves some to tears of joy is in the eyes of others only a green thing that stands in the way.

– William Blake

Contents

1 Type Inference and Deforestation **7**
- 1.1 Haskell Notation . . . 7
  - 1.1.1 Lambda Notation . . . 7
  - 1.1.2 List Notation . . . 8
  - 1.1.3 Types . . . 9
  - 1.1.4 Higher-Order Functions . . . 10
  - 1.1.5 Comments . . . 10
- 1.2 Modular and Monolithic Programs . . . 11
- 1.3 Deforestation In Theory . . . 13
- 1.4 Deforestation In Practice . . . 13
  - 1.4.1 Wadler's Deforestation Algorithm . . . 13
  - 1.4.2 Shortcut Deforestation . . . 16
  - 1.4.3 Warm Fusion . . . 19
- 1.5 The Type Inference Approach . . . 21
- 1.6 Implementing the Type Inference Approach . . . 22

2 Translating Between Haskell and F **25**
- 2.1 Core and the GHC Front-End . . . 25
- 2.2 Differences Between Core and F . . . 26
- 2.3 The Core-to-F Translator . . . 30
- 2.4 The F-to-Haskell Translator . . . 32

3 Tools for F **39**
- 3.1 The F Typechecker . . . 39
- 3.2 The F Interpreter . . . 44

4 Shortcut Deforestation **51**
- 4.1 The List Abstraction Algorithm . . . 51
- 4.2 Postprocessing Steps . . . 52
- 4.3 Applying the Shortcut Rule . . . 55

5 Conclusions and Future Work **57**
- 5.1 Working Examples . . . 57
- 5.2 Future Work . . . 58
  - 5.2.1 Type-Based Cloning . . . 58

5.2.2 Beyond Lists . . . 59
5.2.3 Benchmarking . . . 59
5.2.4 Integration with GHC . . . 59

A Haskell Code 63
A.1 List of Modules . . . 63
A.2 Main.hs . . . 63
A.3 RunTypeInference.hs . . . 65
A.4 ParseFile.hs . . . 69
A.5 TransCore.hs . . . 71
A.6 Typecheck.hs . . . 74
A.7 Eval.hs . . . 77
A.8 Table.hs . . . 87
A.9 Result.hs . . . 88
A.10 F2Haskell.hs . . . 90
A.11 TypeInference.hs . . . 93
A.12 Core.hs . . . 110
A.13 ThisUtil.hs . . . 119
A.14 ThisUnique.hs . . . 123
A.15 MonadTransformer.hs . . . 125
A.16 CountIO.hs . . . 127
A.17 PrettyCore.hs . . . 128

B Running Test Cases 131

Acknowledgments

I would like to thank the following people:

Olaf Chitil, for doing the work upon which this thesis is based, and for letting me have access to his work,

My advisor, Lyn Turbak, for always being there – literally,

Kate Golder, for the `ParseFile` code,

The Committee on Curriculum and Instruction and the Jerome A. Schiff Foundation, for their financial support,

David Farris, for his emotional support,

My parents, for their various improbable actions leading up to the the even more improbable event of my existence,

Lari Ranbom, for her constant reminders to write my thesis, and for helpful suggestions such as "You should write it in the form of a romance novel,"

Marianne Terrot, Emily Braunstein, Katie Hayes, and Sheree Schrager, for suggesting many excellent thesis titles, all of which I promptly rejected,

Alexandra Dunne-Bryant, for driving the Kinko's Express,

Alan Turing, Alonzo Church, John von Neumann, John Backus, John McCarthy, Richard Stallman, Donald Knuth, Leslie Lamport, Linus Torvalds, Paul Hudak, Simon Peyton Jones, and Phil Wadler, without whom this thesis would not be possible,

And finally, the many trees – both physical and virtual – who sacrificed their lives for this thesis.

Chapter 1

Type Inference and Deforestation

Modularity is a good technique for designing programs that are easy to understand and maintain. However, modularity can make programs run less efficiently, in part due to the creation of intermediate data structures. Deforestation is a program transformation that eliminates some intermediate data structures. Various techniques for deforestation have been proposed, but many of them rely on the input program being in a specific, restricted form. The type inference approach to deforestation lessens these restrictions.

In his Ph.D thesis, Olaf Chitil described type-inference-based deforestation [Chi00]. He also developed a prototype implementation of the core part of it, the list abstraction algorithm. In this paper, I describe how I implemented some of the remaining stages of type-inference-based deforestation, and adapted it to work on Haskell programs and produce Haskell programs.

1.1 Haskell Notation

The type-inference-based deforestation system is implemented in the Haskell programming language, and I will use Haskell for the examples in this paper. This section explains certain features of Haskell, for those unfamiliar with the language. Readers who are familiar with Haskell may skip it.

1.1.1 Lambda Notation

The notation (\ x_1 x_2 ... x_n -> <body>) denotes a function that takes n arguments, x_1 through x_n, and returns <body>. The \ is Haskell's representation of a λ, which is the symbol conventionally used to introduce a function in functional languages.

In Haskell, functions can be *curried*. We say that a function of n arguments is curried if, given m arguments (where $m < n$), it returns a function that takes $n - m$ arguments. For example, suppose we wanted to define a curried function to add up two numbers:

```
add x y = x + y
```

The call `(add 3)` would give us back a function that takes one argument, `y`, and returns `(3 + y)`. The call `((add 3) 2)` would be equivalent to `(add 3 2)`, which evaluates to 5.

We say that an application to m arguments of a curried function that takes n arguments (in total) is *saturated* if $n = m$. For the above example, `(add 3 2)` would be saturated, but `(add 2)` would not be.

A function that takes multiple arguments can do so either via currying, or via *tupling*. A tuple is a way to group together several elements (a tuple of n elements is known as an n-tuple). Tuples provide another way for a function to take more than one argument, or to return more than one result: for example, the following function takes a 2-tuple (that is, a pair) of numbers and returns a 2-tuple of their sum and product:

```
sumandproduct (x, y) = (x + y, x * y)
```

The call `(sumandproduct (5, 42))` would return the tuple `(47, 210)`.

1.1.2 List Notation

A list is either the empty list, written as `[]`, or the result of *cons*ing an element of type α onto a list whose elements are all of type α, using the infix `:` operator (cons). For example, `(1:(2:[]))` is a list of the integers one and two. The infix operator `:` can be used as a prefix operator by writing it as `(:)`. We say that `(:)` and `nil` are the *constructors* for the `List` datatype.

The list (x_0:(x_1: ... (x_n:[]))) can also be written as [x_0, x_1, ..., x_n], or visually represented as a tree of constructor applications (see Figure 1.1).

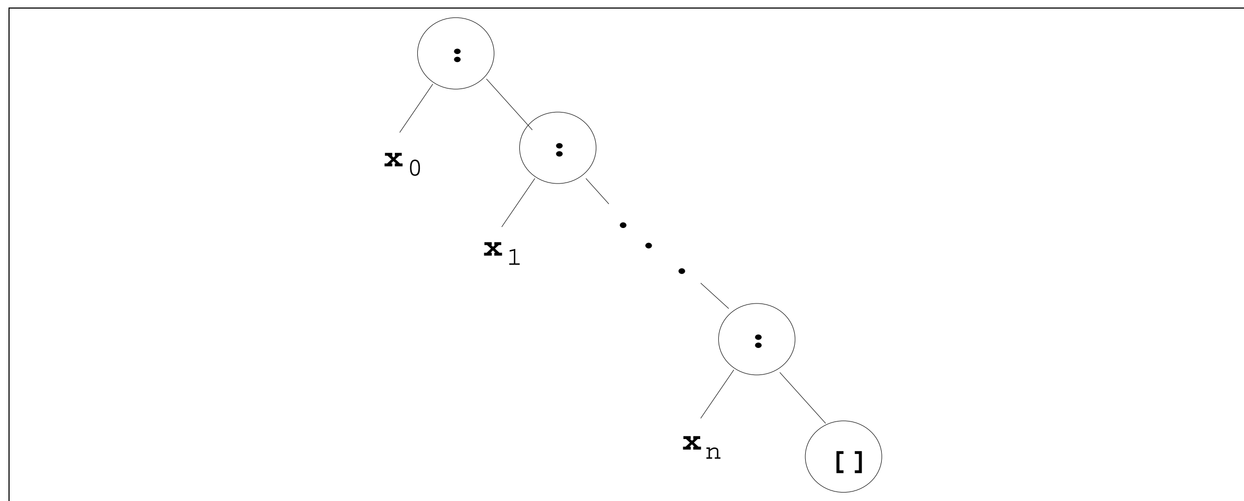

Figure 1.1: An sample list

Here are two examples of functions that consume and produce lists:[1]

[1]Haskell provides a shorthand notation `[m..n]` for `(fromTo m n)`.

```
fromTo x y = (if (x > y) then
                []
              else
                (x:(fromTo (x + 1) y)))

sum l = (case l of
          [] -> 0
          (x:xs) -> (x + (sum xs)))
```

The `sum` function is defined using pattern-matching on its list argument. The `case` statement says that if the input list, `l`, is the empty list, then return 0; otherwise, if the list consists of an element `x` consed onto another list `xs`, then return `x` plus the sum of `xs`.

1.1.3 Types

Haskell requires every expression to have a well-defined type. The type of a curried function that takes n arguments, whose types are t_1, t_2, ..., t_n, and that returns a result of type t, is written as $(t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t)$. The $\rightarrow$ operator associates to the right. For example, the type of the `+` function, that takes two `Int`s and returns an `Int`, is `(Int → Int → Int)`. The right-associativity of the $\rightarrow$ operator means that given an `Int`, `+` will return a function of type `(Int → Int)`, and given two `Int`s, `+` will return an `Int`.

The type of a list whose elements have type t is written as `[t]`, and the type of an n-tuple whose elements have types $t_1 \ldots t_n$ is written as $(t_1, \ldots, t_n)$.

Another data type is the *record* type. When defining a new type, it can be declared as a record: an object that, like a tuple, bundles together several values (fields), but unlike a tuple, associates names with these fields. For example, we could define a type to represent a person:

```
data Person = Person { name :: String,
                       age :: Int
                     }
```

To create a new element of this type, then refer to its fields, we could write:

```
let bob = Person {name="Bob", age=42 }
in
  (name bob) ++ " is " ++ (show (age bob))
```

Some types in Haskell are polymorphic – that is, their definitions include type variables, which can range over all possible types. For example, the type of the `map` function (explained below) is $(\forall\, \alpha\, \beta\, .\ ((\alpha \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]))$. The $\forall$ is read "for all", and this type would be written in Haskell as `((a -> b -> b) -> [a] -> [b])` (the $\forall$ is implicit).

Haskell does not require users to write explicit types for the functions they define, but it is possible to do so, using the `::` (pronounced "has type") operator to separate a function name and its type. Here are type declarations for the `fromTo` and `sum` functions from the previous section:

```
fromTo :: Int -> Int -> [Int]
sum :: [Int] -> Int
```

1.1.4 Higher-Order Functions

Functional languages usually provide certain higher-order functions – functions which take other functions as arguments. Two of the most common ones are `map` and `foldr`.

The `map` function takes a function `f`, of type $(\alpha \to \beta)$, and a list `l`, of type $[\alpha]$, and returns a list of type $[\beta]$, consisting of the results of applying `f` to each element of `l`. For example, if `f` were `square`, which has type `(Int` $\to$ `Int)`, and `l` were `[3, 4, 5]`, which has type `[Int]`, the resulting list would also have type `[Int]`. Figure 1.2 illustrates how `(map square [3, 4, 5])` is computed.

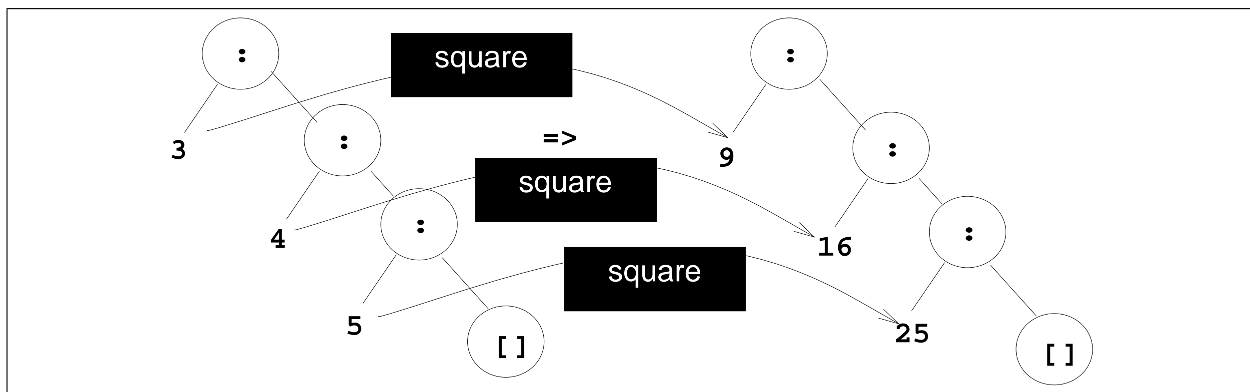

Figure 1.2: `(map square [3, 4, 5])`

The function `foldr` takes three arguments: `f`, whose type is `(`$\alpha \to \beta \to \beta$`)`, `start`, whose type is β, and a list `l`, whose type is `[`α`]`, and returns a result of type β. For example, if `f` were `+`, which has type `(Int` $\to$ `Int` $\to$ `Int)`, `start` were `0`, which has type `Int`, and `l` were `[3, 4, 5]`, the result would have type `Int`.

`(foldr f start l)` starts by combining the last element in `l` with `start`, using `f`, and continues from right to left, combinining each element in `l` with the accumulated result. Essentially, it replaces the constructors `(:)` and `[]` in `l` with `f` and `start`, and evaluates the result. Figure 1.3 illustrates how `(foldr + 0 [3, 4, 5])` is computed.

1.1.5 Comments

In Haskell, a comment begins with two dashes and continues to the end of a line:

```
-- No comment.
```

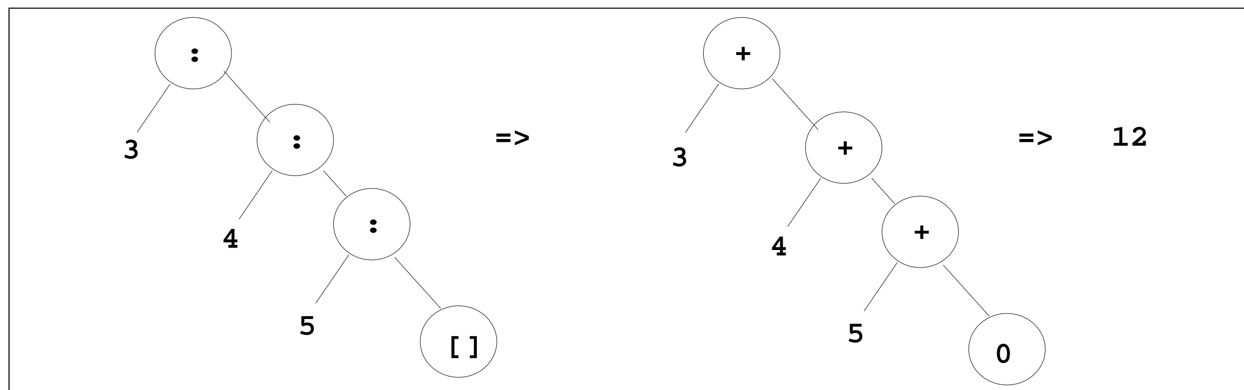

Figure 1.3: (foldr + 0 [3, 4, 5])

1.2 Modular and Monolithic Programs

A modular program is one that is defined in terms of many smaller parts, called *modules*. The division of a program into modules is determined by the natural decomposition of the problem the program solves, the potential for reusing modules in other programs, and the presence or absence of any pre-existing modules that can be called upon. A monolithic program is the opposite of a modular program: it is defined as a single, indivisible unit, with little or no potential for reusing parts of it or using existing parts from other programs.

For example, consider two programs to compute the sum of squares of the integers between m and n. In Haskell notation, here are two possible definitions:

```
sos m n = (foldr + 0 (map square (fromTo m n)))

sos m n = if (m > n) then
            0
          else
            ((square m) + (sos (m + 1) n))
```

The first definition is more concise, because it uses the higher-order functions `foldr` and `map`, which encapsulate the recursion expressed explicitly in the second definition. However, the first definition creates two intermediate lists – `(fromTo m n)` and `(map square (fromTo m n))` – whereas the second definition creates no lists. Figure 1.4 shows the computation of `(sos 3 5)` for the first definition of `sos`. Compare this to computation of the second definition of `sos`, depicted in Figure 1.5.

Modular definitions tend to create intermediate data structures like those shown in Figure 1.4. Functional programmers embrace this style of programming, in which different parts of a program use lists and other tree-shaped data structures to communicate with each other. As John Hughes puts it, such data structures are the "glue" that holds the modular parts of a program together [Hug89]. But creating these structures uses time and space which would not be necessary for executing the equivalent monolithic definitions. Function calls themselves add extra overhead as well. Although modularity makes programs easier for humans to understand, it makes them harder (more expensive, in terms of time and space) for computers to execute, for exactly the same reasons.

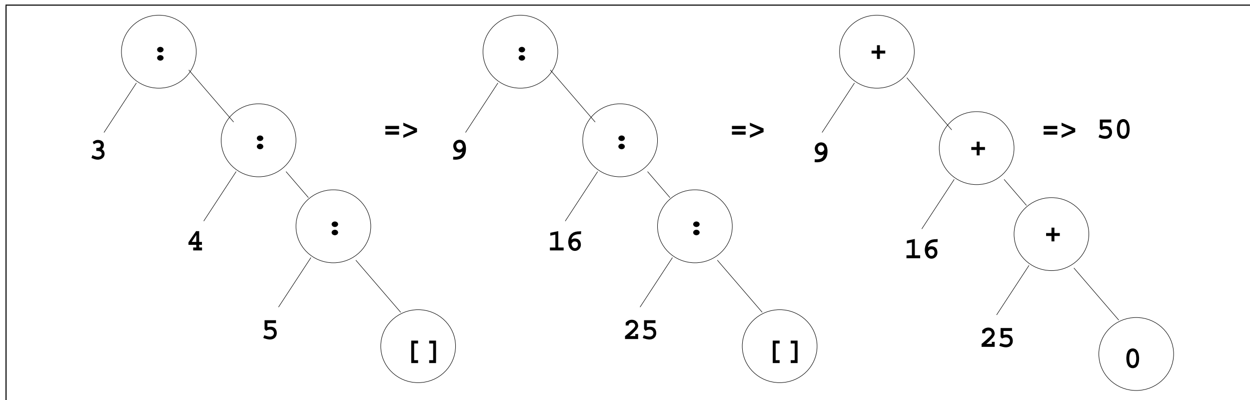

Figure 1.4: `(sos 3 5)` – modular definition

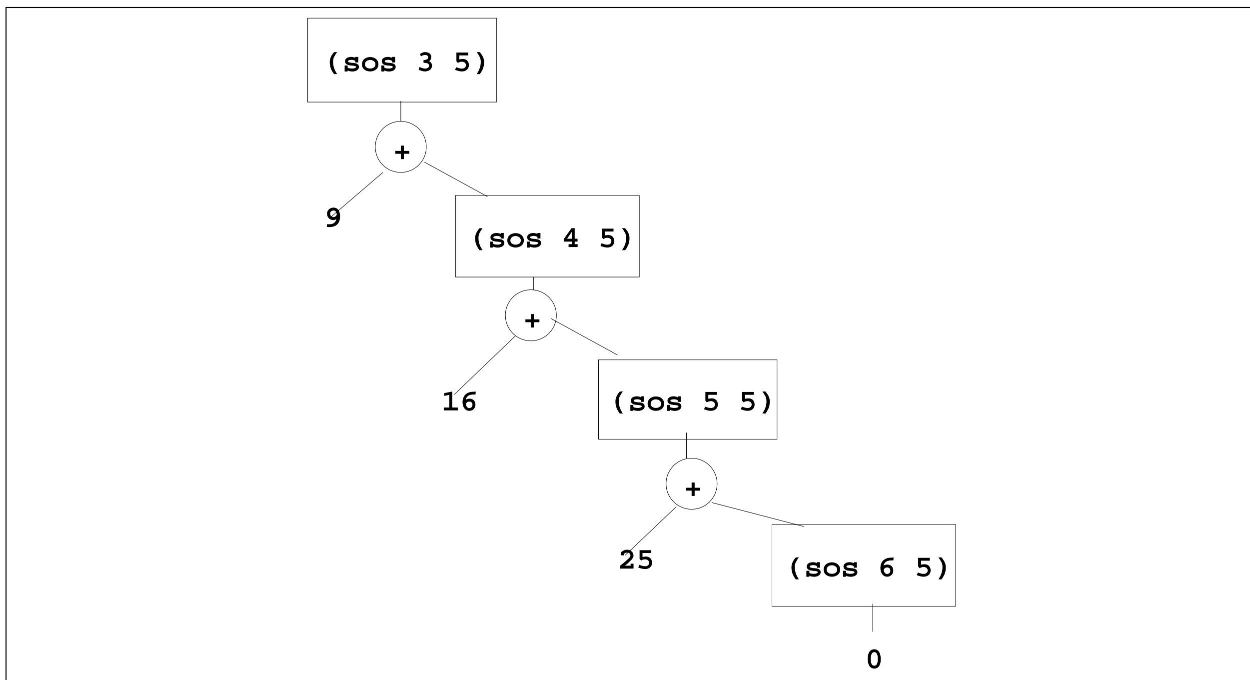

Figure 1.5: `(sos 3 5)` – monolithic definition

1.3 Deforestation In Theory

The goal of deforestation is to reduce this tradeoff between efficiency and modularity. Deforestation is a program transformation to eliminate certain intermediate data structures, turning programs like the first definition of `sos` into programs like the second definition. The idea is that an algorithm can determine which data structures are actually essential to performing the computation at hand, and which are just there to faciliate modularity – call the second kind *virtual* data structures. Then, it can rewrite the program to fuse each composition of a list-consuming function with a list-producing function into a single function, eliminating the virtual lists.

1.4 Deforestation In Practice

1.4.1 Wadler's Deforestation Algorithm

Deforestation was first proposed by Philip Wadler, who coined the term "deforestation" for a process that removes intermediate data structures such as lists and trees. He introduced an algorithm for fusing together compositions of functions, which assumed that the functions themselves created no intermediate data structures (other than those produced and consumed by the functions) [Wad90]. This algorithm revolved around unfolding function calls in an expression with the goal of transforming it into a `case` expression whose scrutinee is a data constructor application – i.e., of this form:[2]

```
case (C arg_1 ... arg_n) of
  p_1 -> e_1
    ...
  p_n -> e_n
```

where one of the p_i's matches (`C` arg_1 ... arg_n). It uses a set of rewrite rules, of which the central one is the "case-on-constructor" rule. For example, this rule would reduce the following expression:

```
case (C e1 e2 e3) of
 A arg1 arg2 -> r1
 B arg1 arg2 arg3 -> r2
 C arg1 arg2 arg3 -> r3
```

to `r3`, the right-hand-side of the clause that matches the scrutinee, with every instance of `arg1`, `arg2`, and `arg3` replaced with `e1`, `e2`, and `e3`, respectively. As a result, the data structure represented by `(C e1 e2 e3)` is never computed.

Here is an example of Wadler's algorithm in action. Consider the modular definition of `sos` from before:

```
sos m n = (foldr + 0 (map square (fromTo m n)))
```

[2] The notation `arg_n` in code is equivalent to arg_n in text.

For Wadler's algorithm to work, this definition has to be written in terms of specialized versions of `map` and `foldr` (this step isn't hard to do automatically):

```
mapsquare l = (case l of
                [] -> []
                (x:xs) -> (square x):(mapsquare xs))

sum l = (case l of
           [] -> 0
           (y:ys) -> (y + sum ys))

fromTo x y = (case (x > y) of
               True -> []
               False -> (x:(fromTo (x + 1) y)))

sos m n = (sum (mapsquare (fromTo m n)))
```

As a first step, the deforestation algorithm clones and β-reduces[3] the definitions of `sum`, `mapsquare`, and `fromTo`:

```
sos m n = (case
             (case
               (case (m > n) of
                 True -> []
                 False -> (m:(fromTo (m + 1) n)))) of
               [] -> []
               (x:xs) -> (square x):(mapsquare xs))) of
            [] -> 0
            (y:ys) -> (y + sum ys))
```

Next, it distributes the middle `case` statement:

```
sos m n = (case
            (case (m > n) of
              True -> (case [] of
                        [] -> []
                        (x:xs) -> (square x):(mapsquare xs))
              False -> (case (m:(fromTo (m + 1) n)) of
                         [] -> []
                         (x:xs) -> (square x):(mapsquare xs))) of
             [] -> 0
             (y:ys) -> (y + sum ys))
```

We can simplify the two innermost `case` statements:

[3] To β-reduce a function application, $((\lambda param_1 \ldots param_n \rightarrow body) arg_1 \ldots arg_n)$, is to substitute $arg_1 \ldots arg_n$ for $param_1 \ldots param_n$ in $body$, resulting in a new expression $body'$, and then replace the original function application with $body'$.

```
sos m n = (case
            (case (m > n) of
              True -> []
              False -> (square m):
                       (mapsquare (fromTo (m + 1) n))) of
            [] -> 0
            (y:ys) -> (y + sum ys))
```

and distribute what is now the outermost `case` statement:

```
sos m n = (case (m > n) of
            True -> (case [] of
                     [] -> 0
                     (y:ys) -> (y + sum ys))
            False -> (case (square m):
                           (mapsquare (fromTo (m + 1) n)) of
                       [] -> 0
                       (y:ys) -> (y + sum ys)))
```

Simplifying as before, we get:

```
sos m n = (case (m > n) of
            True -> 0
            False -> (square m) +
                     (sum (mapsquare (fromTo (m + 1) n))))
```

This expression contains a renaming of the original expression: `(sum (mapsquare (fromTo (m + 1) n)))`. If we kept unfolding function calls at this point, we would go into an infinite regress. Instead, we observe that the problematic expression – `(sum (mapsquare (fromTo (m + 1) n)))` – is exactly the expression we just derived, with `(m + 1)` substituted for `m` – in other words, it is a call to the original function, `sos`. We take advantage of this and replace the renamed expression with a recursive call to `sos`:

```
sos m n = (case (m > n) of
             True -> 0
             (x:xs) -> (square m) + (sos (m + 1) n))
```

We have successfully derived a definition for `sos` that creates no intermediate data structures.

It turns out that in the presence of recursive function definitions, Wadler's algorithm as stated may not terminate. Wadler solved this problem by requiring all individual function definitions to be in "treeless form" – essentially, to create no intermediate data structures. He then showed that any composition of treeless functions could be fused into a single function definition by a terminating sequence of rewrite steps (like those shown above).

1.4.2 Shortcut Deforestation

Wadler's algorithm was far too restrictive to be applicable to real programs, so Andrew Gill, John Launchbury, and Simon Peyton Jones later introduced shortcut deforestation, which handles a different range of programs [GLP93, Gil96].

Shortcut deforestation centers around applying the following rule, known as the shortcut rule.

```
foldr k z (build g) = g k z
```

Figure 1.6: The shortcut rule

Here, `g` is a list-valued expression that has been parameterized over its constructors. That is, the list `[1, 2, 3]` would be written as `(\ c n -> (c 1 (c 2 (c 3 n))))`. `build` is a function that takes such an abstract list and applies it to the list constructors `(:)` and `[]` (cons and nil) – see Figure 1.7. The shortcut rule works by applying the abstract list to the arguments of `foldr`. The insight here is that `foldr` works by uniformly replacing the constructors in a list with other functions – see Figure 1.8. Likewise, `build` replaces placeholders which mark where the constructors should go in a list with specific constructors, `(:)` and `[]` – see Figure 1.7. So instead of constructing the list `build g`, we can directly replace the placeholders with the functional arguments of `foldr`, shown in Figure 1.9.

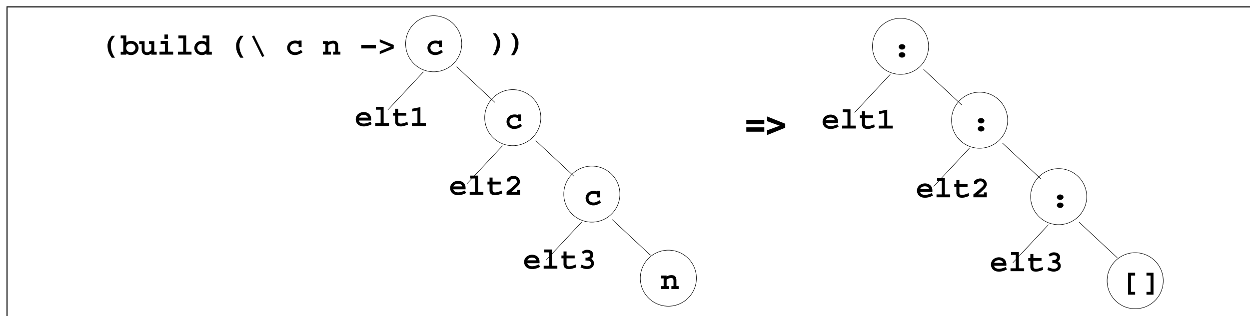

Figure 1.7: Definition of `build`

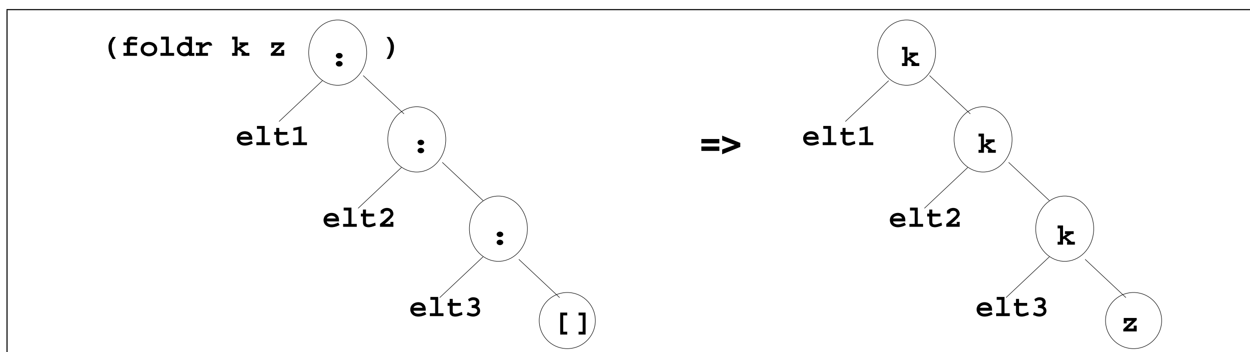

Figure 1.8: Definition of `foldr`

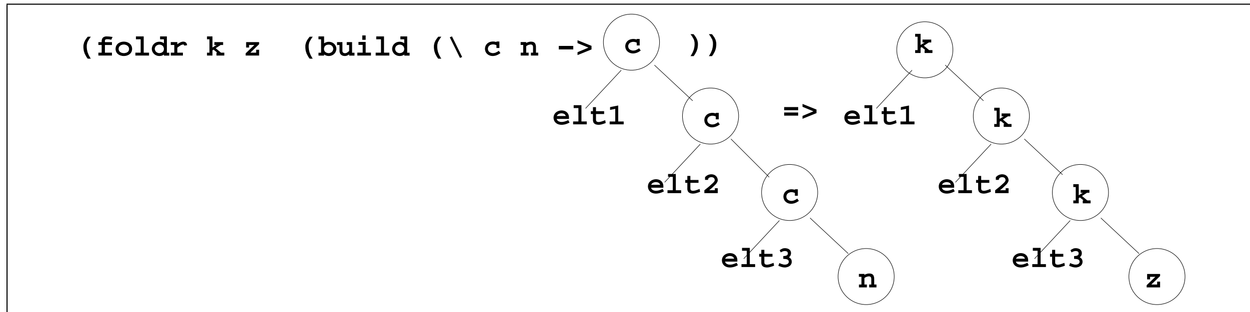

Figure 1.9: The shortcut rule

The list is eliminated when we insert the function arguments `k` and `z` directly in place of the list's constructors, rather than constructing the list and then combining its elements with `k` and `z`. For the shortcut rule to deforest a program, all list-producing functions must be written in terms of `build` and all list-consuming functions must be written in terms of `foldr`.

As an example, consider applying the shortcut rule to the modular definition of `sos`, from above. We first have to write the definition using `foldr` and `build`, and clone the definitions of any functions that use `foldr` and `build`:

```
sos i j = (foldr + 0
            (build
               -----------
               (\ c n ->
                    (foldr (\ x rest -> (c (square x) rest)) n
                       (build
                        -- ************
                        (\ c' n' ->
                          (let fromTo x y =
                                 (if (x == y) then
                                    n'
                                  else (c' x (fromTo (x + 1) y)))
                          in
                          (fromTo i j)))
                        -- ************
                     )))
              -------------
            ))
```

The portion of the program between the dashed lines represents `(map square (fromTo m n))`, and the portion between the starred lines represents `(fromTo m n)`. Applying the shortcut rule to the innermost `foldr`/`build` pair first, we get:

```
sos i j = (foldr + 0
            (build (\ c n ->
                      ((\ c' n' ->
                          (let fromTo x y =
                            (if (x == y) then
                               n'
                             else
                               (c' x (fromTo (x + 1) y)))
                          in
                          (fromTo i j)))
                        (\ x rest -> (c (square x) rest))
                        n))))
```

After performing β-reduction on the application of `(\ c' n' -> ...)`, and then on the application of `(\ x rest -> (c (square x) rest))`, we have:

```
sos i j = (foldr + 0
            (build (\ c n ->
                     (let fromTo x y =
                        (if (x == y) then
                           n
                         else
                           (c (square x) (fromTo (x + 1) y)))
                      in
                        (fromTo i j)))))
```

Applying the shortcut rule to the remaining `foldr`/`build` pair, we get:

```
sos i j = ((\ c n ->
             (let fromTo x y =
                (if (x == y) then
                   n
                 else
                   (c (square x) (fromTo (x + 1) y)))
              in
                (fromTo i j)))
           +
           0)
```

After β-reduction, we have:

```
sos i j = (let fromTo x y = (if (x == y) then
                               0
                             else
                               ((square x) + (fromTo (x + 1) y)))
           in
           (fromTo i j))
```

This definition is equivalent to the original one, but creates no intermediate lists.

The problem with the shortcut method is that it is too restrictive. Although `foldr` is commonly used in functional programs anyway, `build` is not, and merely exists to enable deforestation. Requiring programmers to write code in the awkward `foldr`/`build` form defeats the purpose of deforestation, which is to allow programmers to write easy-to-understand code and translate it into efficient code.

1.4.3 Warm Fusion

Warm fusion, developed by John Launchbury and Tim Sheard, is an attempt to increase the applicability of shortcut deforestation by automatically translating recursive definitions into `foldr`/`build` form [LS95]. The idea is to apply two transformation rules, *buildify* and *cataify* ("cata" is another name for `foldr`) to automatically rewrite any functions which uniformly produce lists in terms of `build`, and rewrite any functions which uniformly consume lists in terms of `foldr`. Then the shortcut rule can be applied [Ném00].

The *buildify* rule works by first transforming an expression `e` which produces a list into `(build (\ c n -> (foldr c n e)))`. This expression is clearly equivalent to the original: the expression `(\ c n -> (foldr c n e))` abstracts over the list constructors in `e`, then applying `build` substitutes `(:)` and `[]` back in. But now, the `foldr` can be pushed inside the expression `e`, which may expose opportunities to perform the shortcut rule. For example, consider applying *buildify* to the definition of `map`. We start with:

```
map f l = (case l of
            [] -> []
            (x:xs) -> (f x):(map f xs))
```

Wrapping a `build`/`foldr` around the definition, we get:

```
map f l = build (\ c n ->
                    (foldr c n
                      (case l of
                        [] -> []
                        (x:xs) -> (f x):(map f xs))
```

Pushing the `foldr` into the right-hand sides of the `case` clauses, we get:

```
map f l = build (\ c n ->
                      (case l of
                        [] -> (foldr c n [])
                        (x:xs) -> (foldr c n (f x):(map f xs))))
```

Now we can simplify the right-hand sides of the *case* clauses:

```
map f l = build (\ c n ->
                      (case l of
                        [] -> n
                        (x:xs) -> (c (f x) (foldr c n (map f xs)))
```

The right-hand side of the second clause still needs to be simplified further, since we would like the `foldr` to be applied to a `build`. If we cloned the definition of `map` here, the argument to `foldr` would indeed be an application of `build`, but then we could clone the recursive call to `map` again – how would we know when to stop? To prevent this infinite regress, we split the definition of `map` into a worker and a wrapper:

```
map f l = (build (mapWorker f l))
mapWorker f l c n = (case l of
                       [] -> n
                       (x:xs) -> (c (f x) (foldr c n (map f xs))))
```

Now that the definition of `map` is non-recursive, we can safely clone it:

```
map f l = (build (mapWorker f l))
mapWorker f l c n = (case l of
                      [] -> n
                      (x:xs) -> (c (f x) (foldr c n
                                    (build (mapWorker f xs)))))
```

Simplifying the `foldr` application, we have:

```
map f l = (build (mapWorker f l))
mapWorker f l c n = (case l of
                      [] -> n
                      (x:xs) -> (c (f x) (mapWorker f xs c n)))
```

We have successfully transformed the definition of `map` into `build` form.

What about *cataify*? We want to take an expression `e` that consumes a list, and redefine it in terms of `foldr`. The idea is that if we knew the value of `e` when the input list is `[]` – call it e_{nil} – and the value of `e` when the input list is `(x:xs)` for some `x` and `xs` – call it e_{cons} – we could rewrite e as `(`λ `xs -> (foldr` e_{cons} e_{nil} `xs))`. We determine what e_{cons} and e_{nil} are by applying `e` to `[]` and to `(x:xs)`. For example, consider applying *cataify* to the definition of `and`, the function that takes a list of booleans $[b_0, b_1, \ldots, b_n]$ and returns (b_0 && b_1 && ... && b_n), where && is the primitive *and* operator (logical conjunction). We start with the definition of `and`:

```
and l = (case l of
           [] -> True
           (x:xs) -> (x && (and xs)))
```

We apply `and` to `[]` and to `(x:xs)` (where `x` is some variable of boolean type, and `xs` is some variable of boolean list type) to get and_{nil} and and_{cons}:

```
and_nil = and [] = True
and_cons = and (x:xs) = (x && (and xs))
```

The definition of `and_cons`, `(x && (and xs))` can be expressed as the function `(\ x xs -> x && xs)`, which has the right type to be the first argument to `foldr`. This may seem like magic, but is dictated by certain simple rules [Ném00]. Now that we know the values of and_{nil} and and_{cons}, we can rewrite `and` in terms of `foldr`:

```
and l = (foldr (\ x xs -> x && xs) True l)
```

We have successfully rewritten `and` in terms of `foldr`.

As is clear even from these trivial examples, warm fusion is complicated. It is computationally expensive and difficult to implement. The idea of automatically deriving `build`s is a good one, though, and we can improve on warm fusion by focusing on deriving `build`s. The use of `foldr` is not problematic, since it commonly occurs in functional programs in any case.

1.5 The Type Inference Approach

In 1999, Olaf Chitil introduced the type inference method for deforestation. Unlike the shortcut method, the type inference method does not require the programmer to use explicit `build`s; instead, it automatically infers where `build` should be applied, and then applies the shortcut rule [Chi99, Chi00]. The core of this method is the list abstraction algorithm, which takes a list-producing expression and returns an abstracted version of the expression, which can then be passed as an argument to `build`. To illustrate this, we will return to the sum-of-squares example. To refresh your memory, the definition of `sos` is as follows:

```
sos m n = (foldr + 0 (map square (fromTo m n)))
```

We first need to clone the definitions of `map` and `fromTo` (a preliminary stage of type-inference-based deforestation, which I do not discuss here, can do this automatically):

```
sos x y = (foldr + 0
            (foldr (\ x rest -> (square x):rest)
                   []
                   (let fromTo x y = (if (x == y) then
                                         []
                                       else
                                         (x:(fromTo (x + 1) y)))
                    in
                     (fromTo x y))))
```

The list abstraction algorithm first abstracts the argument of the inner `foldr` over its list constructors, replacing each instance of `:` or `[]` with variables c and n, respectively:

```
sos x y = (foldr + 0
            (foldr (\ x rest -> (square x):rest)
                   []
                   (build (\ c n ->
                          (let fromTo x y =
                             (if (x == y) then
                                n
                              else
                                (c x (fromTo (x + 1) y)))
                           in
                             (fromTo x y))))
```

Next, it processes the argument of the outer `foldr`:

```
sos x y = (foldr + 0
            (build (\ c' n' ->
              (foldr (\ x rest -> (c' (square x) rest))
                     n'
                     (build (\ c n ->
                            (let fromTo x y =
                              (if (x == y) then
                                 n
                               else
                                 (c x (fromTo (x + 1) y)))
                             in
                               (fromTo x y)))))))
```

Now we can apply the shortcut rule to both `foldr`/`build` pairs, just as in the shortcut rule example. Notice that the steps are very similar to those of shortcut deforestation. The difference is that here, the placement of `build`s is inferred automatically, rather than determined by the programmer.

1.6 Implementing the Type Inference Approach

Chitil developed a prototype implementation of the list abstraction algorithm. The prototype accepts programs in F, a simple functional language based on Core, the intermediate language used by the Glasgow Haskell Compiler (GHC). I made the following changes and additions to the prototype:

- I modified the prototype to handle lists of arbitrary type (Chitil's version only handled lists of booleans.)
- I developed a program to translate Haskell programs into F. Initially, it was necessary to write programs in the abstract syntax of F in order to run the list abstraction algorithm, since there was no parser for F. This translator greatly eased writing test programs.

- I developed a typechecker and interpreter for F, in order to validate the results produced by the list abstraction algorithm. These tools also facilitated testing.
- I implemented the shortcut rule for F.
- I developed a program to translate F programs back into Haskell, so that the results of deforestation can be compiled in GHC and compared to the results of compiling the original programs.

Now that these tools exist, with a little more work it should be relatively easy to run the type inference algorithm on standard benchmarks and compare the resulting performance gains to those resulting from other methods for deforestation.

The remainder of this paper describes the implementations of the tools I developed. Chapter 2 describes the Core-to-F translator, along with the process of utilizing the GHC front-end to get code from Haskell into Core, and the F-to-Haskell translator.. Chapter 3 describes the typechecker and interpreter for F, which are useful for validating the results of the type inference algorithm. Chapter 4 describes the existing list abstraction algorithm and my implementation of the shortcut rule for F. Chapter 5 indicates some possible directions in which to expand on my work.

Figure 1.10 shows the stages of my extended version of type-inference-based deforestation. Stages contributed by me are indicated with **bold** borders.

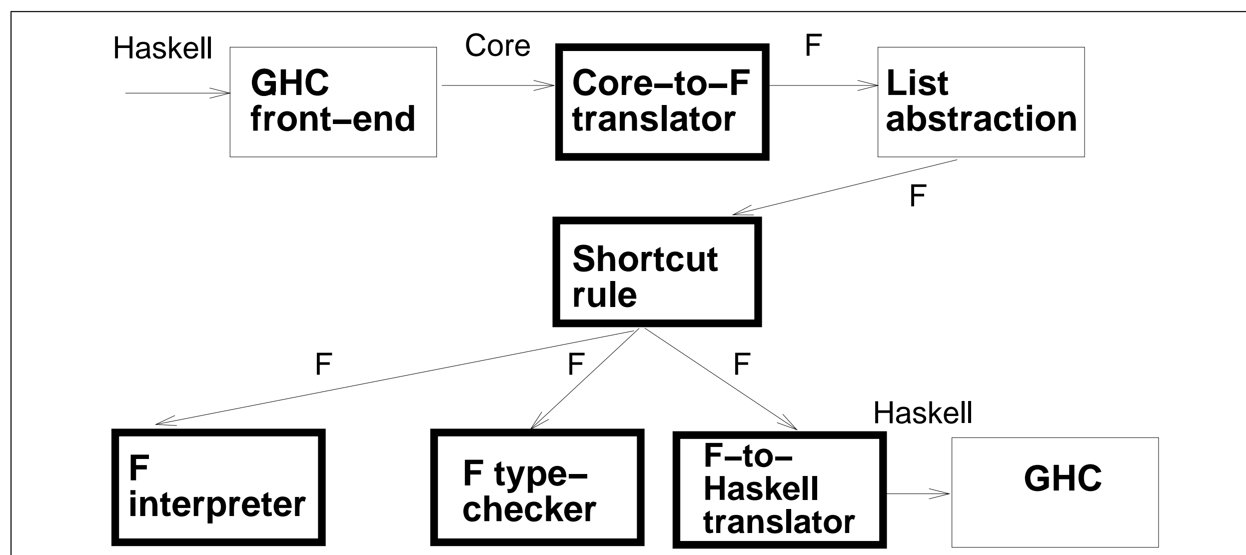

Figure 1.10: The stages of type-inference-based deforestation

Chapter 2

Translating Between Haskell and F

Chitil's prototype implementation of the type inference algorithm accepts programs written in F, a modified and simplified version of Core. Core is the intermediate language used in the Glasgow Haskell Compiler (GHC). With Chitil's original system, the only way to test the list abstraction algorithm was to write programs in the abstract syntax of F, and read the resulting F programs to verify their correctness. This was extremely tedious and error-prone. To address this issue, I wrote a Haskell-to-F translator, which exploits the GHC front-end to compile a Haskell program into Core, then translates the resulting Core program into F. In addition, I wrote a translator that prints an F abstract syntax tree as a Haskell program, in order to ease verifying the results of the list abstraction algorithm. This chapter describes these translators..

2.1 Core and the GHC Front-End

The GHC front-end is composed of several stages: the parser, the renamer, the typechecker, and the desugarer. The parser converts the text of a Haskell program to a Haskell abstract syntax tree (AST), and the renamer and typechecker each take a Haskell AST and return a Haskell AST. The desugarer takes the renamed and typechecked Haskell AST and returns a Core AST. The process is illustrated in Figure 2.1.

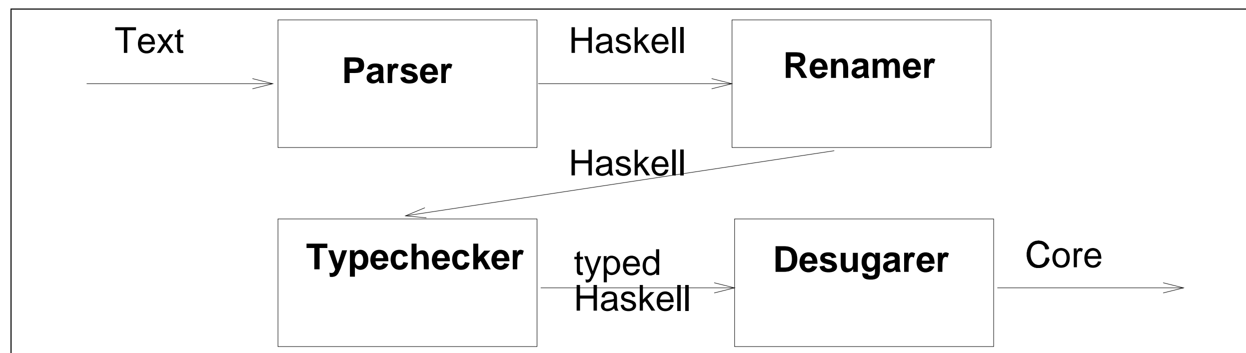

Figure 2.1: The GHC front-end

The syntax of Core is as given by the Haskell datatype definitions shown in Figure 2.2,

with some additional definitions in Figure 2.3 (those definitions which are not particularly enlightening are not shown).

Core expressions are a limited subset of Haskell expressions, and can either be a variable, a literal (either a character, a string, a pointer, an integer, a word, a float, or a double), a function application, a λ abstraction, a `let` expression, a `case` expression, a note, or a type. A note is an expression coupled with a hint about it for the compiler, such as a directive to inline it wherever possible. A type can be a type variable, a type application, a type constructor application, a function type, a note type (analogous to a note expression) or a forAll type.

```
data Expr var -- Expr is polymorphic over the type of identifiers
  = Var   Id                    -- includes data constructors
  | Lit   Literal               -- i.e., integers and characters
  | App   (Expr var) (Arg var)      -- Arg is a synonym for Expr
  | Lam   var (Expr var)
  | Let   (Bind var) (Expr var)
  | Case  (Expr var) var [Alt b]
    -- The variable is bound to the case expression
    -- DEFAULT case must be last, if it occurs at all
  | Note  Note (Expr var)
  | Type  Type

data Type
  = TyVarTy TyVar
  | AppTy Type Type
  | TyConApp TyCon [Type]
  | FunTy Type Type
  | NoteTy TyNote Type
  | ForAllTy TyVar Type
```

Figure 2.2: Syntax of the Core language

2.2 Differences Between Core and F

The syntax of F is shown in Figure 2.4. An F expression can either be a variable, a literal (which can be either a character or an integer), a data constructor application, a function application, a λ abstraction, a `let` expression, a `case` expression, or a type. (The syntax of F could easily be extended to include other basic types, such as strings and floats.) A type can be either a type variable, a type constructor application, a function type, or a for-all type.

Chitil's version of F did not include literals, literal patterns, or the `DEFAULT` pattern. The only literals it supported were booleans, represented by applications of the data constructors `True` and `False` to no arguments. Adding these was the only change to F necessary in order to accurately represent Core.

```
type Id = Var
data Var
  = Var {
        varName    :: Name,
        realUnique :: Int#,        -- Key for fast comparison
        varType    :: Type,
        varDetails :: VarDetails,
          -- Indicates whether this Var is an ordinary
          -- identifier or a type variable
        varInfo    :: IdInfo
          -- Stores various information about this Var
     }

data Name = Name {
                n_sort :: NameSort,     -- What sort of name it is
                n_uniq :: Unique,
                n_occ  :: OccName,      -- Its occurrence name
                n_prov :: Provenance    -- How it was made
             }

data Bind varType = NonRec varType (Expr varType)
            | Rec [(varType, (Expr varType))]
-- either a single binding, or a mutually recursive
-- list of bindings

type Alt varType = (AltCon, [varType], Expr varType)

data AltCon =
   DataAlt DataCon  -- a pattern can either be a data constructor,
 | LitAlt  Literal  -- a literal,
 | DEFAULT          -- or the DEFAULT alternative, which matches any expression
```

Figure 2.3: Auxiliary types for Core syntax

```
data Expr var
  = Var   Id
  | Lit   Literal
  | Con   Con [Arg var]
  | App   (Expr var) (Arg var)
  | Lam   var (Expr var)
  | Let   (Bind var) (Expr var)
  | Case  (Expr var) var [Alt var]
  | Type  Type

type Id = Var

data Var
  = VarG {
        varName    :: Name,
        realUnique :: Unique,
        varType    :: Type,
        varDetails :: VarDetails
      }

data Con = C Name
data Name = Named {
                    n_occ :: OccName,
                    n_uniq :: Unique
        }

data Literal = MachChar Char | MachInt Integer

data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]

type Alt b = (Pat, [b], Expr b)
data Pat = ConPat Con | LitPat Literal | DEFAULT

data Type
  = TyVarTy TyVar
  | TyConApp TyCon [Type]
  | FunTy Type Type
  | ForAllTy TyVar Type
```

Figure 2.4: Syntax of the F language

There are a number of other differences between Core and F. For one, the `varInfo` field of the `Var` type in Core is missing in F, as it is used only for the optimizations GHC performs. The `Name` type, used for the `varName` field of `Var`, includes the fields `n_sort` and `n_prov` in Core which are not present in F's `Name` type; these fields store information about names which F doesn't need. The `Note` expression type is missing in F, as are the `NoteTy` and `AppTy` data types. Case patterns are represented slightly differently, because data constructors are represented differently in the two languages. Core includes many more types of literals than F does, and many more type constructors: the only type constructors available in F are `List`, `Tuple`, `Bool`, `Kind`, `Int`, and `Char` (the last two were added by me; see Chapter 5 for discussion of the possibility of adding more types to F). Finally, F represents data constructor applications differently from function applications, and requires data constructor applications to be saturated (that is, such that the number of arguments is exactly the number the constructor demands). In Core, constructor applications and function applications are superficially indistinguishable.

As an example, consider the simple Haskell expression shown in Figure 2.5 and its corresponding Core AST. (Suppose that `consId`, `nilId`, `trueId`, `falseId`, `caseExpId`, `xId`, and `xsId` are properly defined `Vars`, `nilDataCon` and `consDataCon` are properly defined `DataCons`, and `lit1` is a properly defined `Literal`.)

```
case [1] of
 [] -> True
 (x:xs) -> False
```

Figure 2.5: A simple Haskell expression

```
(Case (App (App (Var consId) (Lit lit1))
        (Var nilId))
  (Var caseExpId)
  [((DataAlt nilDataCon), [], (Var trueId)),
   ((DataAlt consDataCon),
    [xId, xsId],
    (Var falseId))])
```

Compare to this the equivalent F AST (as before, assume that any variable names represent properly defined instances of the appropriate type):

```
(Case (Con consCon [(Lit lit1)])
  (Var caseExpId)
  [((ConPat nilCon), [], (Con trueCon [])),
   ((ConPat consCon),
    [xId, xsId],
    (Con falseCon []))])
```

2.3 The Core-to-F Translator

The following section contains technical details about the Core-to-F translator, and is intended for those who may be extending the system. The casual reader is advised to skip it.

The translation from Core to F is mostly straightforward, except for the handling of data constructors. Since data constructors are represented as variables in Core, many special cases are necessary to translate them correctly. Eventually, it would be desirable to handle constructors in a more general way, so that a wider range of constructors could be easily handled.

The code for the first few cases of the translator is shown in Figure 2.6[1]. The case for `Var` has special cases for the data constructors included in F that have no arguments – currently, `True` and `False`. Similarly, the case for `App` checks whether the operand is a data constructor. This is done using special pattern-matching cases for applications of Core's `cons` and `nil`. An application of `cons` or `nil` is translated into a Core `Con` with the corresponding constructor. `True`, `False`, `cons`, and `nil` are the only data constructors the translator currently handles – any other data constructors are translated into variables. (The correct behavior would actually be to signal an error if an unsupported data constructor is encountered – this would entail only a minor change to the code.)

The remaining cases of the translator, which are straightforward, are shown in Figure 2.7.

Several of the auxiliary functions used by the `trans` function are shown in Figure 2.8. As before, `transAlt` uses special cases in order to properly translate instances of `cons` and `nil` in `Case` patterns. `transName` extracts the only two relevant fields – `occNameString` and `nameUnique` – out of a `Name`, the object that, paired together with a `Type`, represents a `Var`. `transLit` handles the only two types of literals supported in F – integers and characters – and signals an error if a literal of any other type is found.

With the exception of the functions `transTy`, `transCon`, and `transApp`, whose definitions are straightforward, the code in Figures 2.6, 2.7, and 2.8 constitutes the entire Core-to-F translator. The translator returns an equivalent F program for any Core program restricted to the basic data types that F includes (integers, characters, booleans, lists – tuples are included in F but not yet implemented in the translator (it would be simple to add them)). It has been successfully tested on many simple programs which manipulate lists.

As an example of the Core-to-F translator in action, consider the very simple Haskell program shown in Figure 2.9, which maps the identity function onto the constant list `[True]`. I included explicit type annotations in order to make the resulting Core code as simple as possible (without explicit types, the GHC typechecker would have inserted polymorphic types for various functions, making the code more complicated.) The result of compiling this program into Core is shown in Figure 2.10, pretty-printed in a Haskell-like syntax. Although each function appears to have two definitions, these actually correspond to the worker and wrapper of each function, which have different unique identifiers (the Core pretty-printer

[1] For clarity, throughout this document, the module name `Core` is substituted for the module name `CoreSyn`, the actual name of the module that defines the syntax of Core, and the name `F` is substituted for the name `Core`, the actual name of the module that defines the syntax of F. In addition, the name `Expr` is substituted for `CoreExpr`, the name of the type representing expressions in both languages.

```
trans :: Core.Expr -> F.Expr
trans (Core.Lit l)              = (F.Lit (transLit l))
trans (Core.Var id) =
-- special cases for True and False
    case (Id.idFlavour id) of
      (IdInfo.DataConId con) ->
          (if (con == TysWiredIn.trueDataCon) then
           (F.Con F.true [])
           else if (con == TysWiredIn.falseDataCon) then
           (F.Con F.false [])
           else
           (F.Var (transId id)))
      _ -> (F.Var (transId id))
trans a@(Core.App
     (Core.App (Core.App (Core.Var var) ty1) arg1) arg2) =
-- special case for cons
    case (Id.idFlavour var) of
     (IdInfo.DataConId con)     -> handleCons con a
     (IdInfo.DataConWrapId con) -> handleCons con a
     otherwise                  -> (transApp a)
  where handleCons con  a@(Core.App (Core.App
                                      (Core.App
                                       (Core.Var var) ty1)
                                    arg1)
                         arg2) =
         if (con == TysWiredIn.consDataCon) then
           (F.Con F.cons [(trans ty1), (trans arg1), (trans arg2)])
         else
           (transApp a)
trans a@(Core.App (Core.Var var) ty1) =
    case (Id.idFlavour var) of
-- special case for nil
     (IdInfo.DataConId con)      -> handleNil con a
     (IdInfo.DataConWrapId con)  -> handleNil con a
     otherwise                   -> (transApp a)
 where handleNil con a@(Core.App (Core.Var var) ty1) =
         if (con == TysWiredIn.nilDataCon) then
           (F.Con F.nil [(trans ty1)])
         else
           (transApp a)
```

Figure 2.6: The Core-to-F translator – `Lit`, `Var`, and special cases of `Apps`

```
trans a@(Core.App _ _)        = (transApp a)
trans (Core.Lam arg exp)      = (F.Lam (transId arg) (trans exp))
trans (Core.Let binds exp)    = (F.Let (transBinds binds)
                                              (trans exp))
trans (Core.Case exp var alts) = (F.Case (trans exp)
                                     (transId var)
                                     (transAlts alts))
trans (Core.Note _ exp)       = (trans exp) -- discard Note information
trans (Core.Type ty)          = (F.Type (transTy ty))
```

Figure 2.7: The Core-to-F translator – `App`, `Lam`, `Let`, `Case`, `Note`, and `Type`

doesn't print out these unique identifiers, so I marked the wrappers by appending a "W" onto their names.) In the Core code, cons and nil are sometimes written as `:` and `[]`, and sometimes written as `$w:` and `$w[]` – for our purposes, these two different representations are equivalent. The result of translating this program into F is shown in Figure 2.11. Note that the bindings for `main` and `show` are missing – the Core-to-F translator removes these, as their types are not supported by F.

Eventually, it would be desirable to translate the F program returned by the deforestation algorithm back into Core, so that type-inference-based deforestation could be incorporated as an optimization pass into GHC. This would be a nontrivial process, as we would need to find a way of saving and later restoring the information that the Core-to-F translator discards (see Chapter 5).

2.4 The F-to-Haskell Translator

The output of the list abstraction algorithm is an F program. For benchmarking purposes, it would be necessary to introduce the output back into GHC, so that the results of compiling it could be compared with the results of compiling the original program. There are two possible ways to do this. The first method would be to translate the F AST back into an equivalent Core AST, which is hard. This method would require preserving and later restoring the information that the Core-to-F translator discards – how this would be done is unclear. The second method is to translate the F AST into a Haskell program and feed it back into GHC, which is easier. I chose the second method, and wrote an F-to-Haskell translator, which takes an F program and returns a string representing the corresponding Haskell program.

The translator is implemented by the `f2haskell` function, and is fairly straightforward. At the top level, it expects the program to have the form `(Let <binds> <body>)`, and the resulting Haskell program will have the form:

```
main = putStr (show <(f2haskell body)>)

<(f2haskell binds)>
```

```
transAlts :: [Core.CoreAlt] -> [F.CoreAlt]
transAlts alts = (map transAlt alts)

transAlt :: Core.CoreAlt -> F.CoreAlt
transAlt (altcon, args, exp) =
    (case altcon of
     Core.DataAlt dc ->
       (if (dc == TysWiredIn.nilDataCon) then
          (Core.ConPat (Core.nil))
       else (if (dc == TysWiredIn.consDataCon) then
          (Core.ConPat (Core.cons))
       else (if (dc == TysWiredIn.trueDataCon) then
          (Core.ConPat (Core.true))
       else (if (dc == TysWiredIn.falseDataCon) then
          (Core.ConPat (Core.false))
       else (Core.ConPat (Core.C (transName
                                  (DataCon.dataConName dc)))))))
     Core.LitAlt  l  -> F.LitPat (transLit l)
     Core.DEFAULT    -> F.DEFAULT,
     (map transId args),
     (trans exp))

transName :: Name.Name -> F.Name
transName nm = F.mkSysLocalName
                (S# (Unique.u2i (Name.nameUnique nm)))
                (OccName.decode
                 (OccName.occNameString (Name.nameOccName nm)))

transId :: Var.Id -> F.Id
transId v = (F.mkIdVar (transName (Var.idName v))
                       (transTy (Var.idType v)))

transLit :: Literal.Literal -> F.Literal
transLit l = case l of
              Literal.MachChar c -> F.MachChar c
              Literal.MachInt i  -> F.MachInt i
              otherwise          -> error ("TransCore: type of " ++
                                           (show l) ++
                                           " not supported")
```

Figure 2.8: Auxiliary functions for the Core-to-F translator

```
mymap :: (Bool -> Bool) -> [Bool] -> [Bool]
mymap f l = (case l of
               [] -> []
               (x:xs) -> (f x):(mymap f xs))

mapmyid :: [Bool]
mapmyid l = (mymap (\ x -> x) l)

main :: IO()
main = (putStr (show (mapmyid [True])))
```

Figure 2.9: `id` program in Haskell

The code for `f2haskell` is shown in Figure 2.12 and Figure 2.13. (The code for the function `convertBinds`, used at the top level of the translator, is not shown – all it does is remove certain unnecessary bindings which are inserted by GHC.)

The only non-trivial part of the translator is printing out variable names. Since GHC splits each function into a worker and wrapper, where the worker and the wrapper have the same name but different uniques, there may be more than one variable with the same name in an F program that was translated from Core. But we can't print out the name with the unique appended onto it in every case, since if we do this for Prelude functions, the result will not be valid Haskell. So when we translate a variable, we have to check whether or not it represents a Prelude function.

The translator discards any type information – for the programs I have tested, it is not necessary to include this information, but for some programs it may be necessary for the translator to include it. If necessary, it would be simple to modify the translator to print out a type signature for the function being defined before each function definition.

For the Haskell expression in Figure 2.5, the result of the translator (after translating the expression into F) is:

```
main = (let ds = (lit:[])
        in (case ds of
             (x:xs) -> False
             [] -> True))

lit = (id (id 1))
```

(I added the indentation – the translator itself doesn't perform any.)

GHC assigned the names `ds` to the `case` expression, and `lit` to the literal 1, so the translator reflects this. Also, the translator replaces certain functions which are inserted by GHC to convert between types, but which are not available to the user, with the `id` function, which explains the definition of `lit`. (It would be fairly simple to change the translator to delete these functions instead, but this is not currently done.)

```
Rec {
mymap :: ((Bool -> Bool) -> [Bool] -> [Bool])
mymap
  = \ f :: (Bool -> Bool) l :: [Bool] ->
        let {
          ds_d1nv :: [Bool]
          ds_d1nv
            = l
         } in
-- by default, GHC assigns the name "wild" to the case expression
          case ds_d1nv of wild_B1 {
-- the @s denote that : and [] are being applied to the type Bool
              : x xs -> : @ Bool (f x) (mymap f xs); [] -> [] @ Bool
           }
mymapW :: ((Bool -> Bool) -> [Bool] -> [Bool])
mymapW
  = mymap
mapmyid :: ([Bool] -> [Bool])
mapmyid
  = \ l :: [Bool] -> mymap (\ x :: Bool -> x) l
mapmyidW :: ([Bool] -> [Bool])
mapmyidW
  = mapmyid
main :: (IO ())
main
  = putStr (show (mapmyid ($w: @ Bool True ($w[] @ Bool))))
mainW :: (IO ())
mainW
  = main
show :: ([Bool] -> String)
show
  = show @ [Bool] $dShow
$dShow :: {Show [Bool] }
$dShow
  = PrelShow.$fShow[] @ Bool $dShow
$dShow :: {Show Bool }
$dShow
  = PrelShow.$fShowBool
end Rec }
```

Figure 2.10: `id` program in Core

```
let {mymap : (Bool -> Bool) -> [Bool] -> [Bool]
     = \f : Bool -> Bool -> \l : [Bool] ->
         let ds : [Bool] = l : [Bool];
         in case ds : [Bool] of wild
             { Cons x xs ->
                 Cons {Bool
                  (f : Bool -> Bool (x : Bool))
                  (mymap : (Bool -> Bool) -> [Bool] -> [Bool]
                    (f : Bool -> Bool)
                    (xs : [Bool])) };
                Nil -> Nil {Bool } };
     mymapW: (Bool -> Bool) -> [Bool] -> [Bool]
     = mymap : (Bool -> Bool) -> [Bool] -> [Bool];
     mapmyid : [Bool] -> [Bool]
     = \l : [Bool] -> mymap : (Bool -> Bool) -> [Bool] -> [Bool]
                        (\x : Bool -> x : Bool)
                        (l : [Bool]);
     mapmyidW : [Bool] -> [Bool] = mapmyid : [Bool] -> [Bool]; }
  in mapmyidW : [Bool] -> [Bool] (Cons {Bool True (Nil {Bool }) })
```

Figure 2.11: `id` program in F

```
f2haskell (Let binds body) = "main = " ++ (f2haskell' body) ++
                             "\n\n" ++ (concat (convertBinds binds))
f2haskell _ = error $ "f2haskell: Let expected"
f2haskell' (Var v) = var2String v
f2haskell' (Lit (MachChar c)) = "'" ++ [c] ++ "'"
f2haskell' (Lit (MachInt i)) = (show i)
f2haskell' (Con con args) =
  (if (con == cons) then
    (parens (sep ":" (f2haskell' (first args'))
                      (f2haskell' (second args')))
   else if (con == nil) then "[]"
   else if ((con == true) || (con == true')) then "True"
   else if ((con == false) || (con == false')) then "False"
   else
     error $ "bad constructor" ++ (pretty con))
   where (_, args') = span isTypeArg args
f2haskell' (App fun (Type _)) = (f2haskell' fun)
f2haskell' (App fun arg) = (parens (sep " " (f2haskell' fun)
                                            (f2haskell' arg)))
f2haskell' (Lam var body) =
  (case (idType var) of
    (TyConApp Kind _) -> (f2haskell' body)
     _ -> (parens ("\\ " ++ (sep " -> " (var2String var)
                                          (f2haskell' body)))))
f2haskell' (Let binds expr) =
 (parens
   (case flat of
     [] -> (f2haskell' expr)
     _ -> (sep "in "
            ("let " ++
             (curlies (concat ((map (bind2haskell True)
                                     (allbutlast flat))
                                ++ [(bind2haskell False (last flat))]))))
            (f2haskell' expr))))
  where flat = flattenBinds binds
f2haskell' (Case expr _ alts) =
 (parens ("case " ++ (f2haskell' expr) ++ " of\n"
         ++ (concat ((map (alt2haskell True) (allbutlast alts))
         ++ [(alt2haskell False (last alts))]))))
```

Figure 2.12: The F-to-Haskell translator

```
bind2haskell :: Bool -> (Var, Expr) -> String
bind2haskell semi (var, expr) = ((sep " = "
                                    (var2String var)
                                    (f2haskell' expr)) ++
                                   (if semi then
                                    ";\n"
                                    else
                                    "\n\n"))

alt2haskell :: Bool -> (Pat, [Var], Expr) -> String
alt2haskell newline (pat, vars, expr) =
    ((sep " -> "
      (pat2string pat vars)
      (f2haskell' expr)) ++ (if newline then
                             "\n"
                             else
                             ""))

pat2string :: Pat -> [Var] -> String
pat2string (ConPat con) vars = (f2haskell' (Con con (map Var vars)))
pat2string (LitPat lit) _ = (f2haskell' (Lit lit))
pat2string DEFAULT _ = "_"


var2String :: Var -> String
var2String var = (case (preludeLookup var) of
 -- for Prelude functions, don't print the unique
                    Just str -> str
                    Nothing -> (pretty var))

parens :: String -> String
parens s = "(" ++ s ++ ")"

curlies :: String -> String
curlies s = " {" ++ s ++ " }"

sep :: String -> String -> String -> String
sep separator firstString secondString =
    firstString ++ separator ++ secondString
```

Figure 2.13: Auxiliary functions for the F-to-Haskell translator

Chapter 3

Tools for F

Originally, the only way to verify the results of the type inference algorithm was to read through the F programs it produced. This was inadequate for thorough testing, so I developed a typechecker and interpreter for F. These tools can be used to compare the input F program (produced by the Core-to-F translator) with the output F program produced by the list abstraction algorithm, using the typechecker to verify that both are well-typed, and using the interpreter to verify that both programs evaluate to the same value.

Both the typechecker and the interpreter return results in the IO monad – that is, of type `IO (Type)` and `IO (Result)`, respectively. Monads are a feature of Haskell that simulate imperative-style programming features such as input/output – in particular, the only way for a program to display debugging or other information while it runs is to embed the commands that display this information within an expression of type `IO (<ty>)`, for some type `<ty>`. Such expressions usually have the form:

```
do
  name1 <- <expr1>
  name2 <- <expr2>
  [...]
-- either:
  exprn
-- or:
  return(val)
```

The lines $name_i$ `<-` $expr_i$ associate the $name_i$'s with the values of the $expr_i$'s, all of which have type `IO (<ty>)`, for some type `<ty>`. The final line of the `do` expression is either `exprn`, an expression of type `IO (<ty>)`, or `return(val)`, where `val` is an expression of type `<ty>`, rather than `IO (<ty>)`.

3.1 The F Typechecker

Typechecking F is fairly straightforward, except that a special rule (discussed below) is necessary for applications of `build`.

The code for the typechecker is shown in Figure 3.1. The case for typechecking a variable works by retrieving the `idType` field stored within the variable. (Most typecheckers work by passing around a type environment, which maps each variable currently in scope onto its type, but since variables in F store their types within them, the type environment is not necessary here.) The cases for constructor applications and function applications handle type arguments using the `applyTys` and `applyTy` functions (defined elsewhere), which apply a polymorphic type to a list of arguments or a single argument, respectively. Also, the case for function applications includes a special case that checks whether the operator `build` and invokes the function `typecheckBuild` (described below) if so.

The case for `Case` expressions is somewhat more lenient than it should be – for an expression `(Case expr var alts)`, it typechecks `expr` and typechecks the right-hand sides of the elements of `alts`, but does not typecheck the left-hand sides of the alts (the case patterns). The reason why is that in the syntax of F, constructor patterns in a `Case` expression do not include the type arguments for the constructor. Typechecking a `Case` constructor pattern could be accomplished by reconstructing the proper type arguments for the constructor from the types of the variables it is applied to in the pattern, and then typechecking the constructor application corresponding to the pattern. This still would not work for all constructors – for example, `Nil` is polymorphic but has no term arguments. So we would have to define a special case to check for such constructors and assume that their types are the same as the type of the `Case` scrutinee.

The rest of the cases of the typechecker are straightforward.

The auxiliary functions for the typechecker are shown in Figure 3.2. The only one worth commenting on is `typecheckBuild`. The rule for typechecking `(build ty buildArg)` should check that `buildArg` has the type $(\forall\beta.(ty \rightarrow \beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta))$. In reality, it checks something more restrictive than this. The `typecheckBuild` function uses the `==` operator on `Types`, which is defined by declaring that the `Type` datatype derives the `Eq` class. This means that when the program compares two `Types` for equality, it does so by checking whether the types are equal component-wise. Thus, the equality test for types is too strict – for example, `((ForAllTy a [a]) == (ForAllTy b [b]))` evaluates to `False` if `a` and `b` are different type variables, even though the two types are functionally equivalent. So, the `build` rule might appear to be somewhat sketchy: the types `first` (the type of the first argument of the function that is the first argument to the argument of `build`) and `someTy` (the type to which `build` is applied) could be any types. (The other types being tested for equality should all simply be type variables, in which case the equality test works correctly.) But in fact, the rule should be exactly right as long as the typechecker is only applied to F programs that were translated from Haskell. Because of the restrictions of Haskell, it should be the case that `build` is never applied to a polymorphic type – hence, the equality test will hold.

To illustrate how the typechecker works, we will execute it on an example program. Consider the Haskell program (which applies the identity function to a constant list by folding `(:)` and `[]` over it) shown in Figure 3.3, and its equivalent F program (pretty-printed in a Haskell-like syntax) after applying the list abstraction algorithm, shown in Figure 3.4. The F program is an excellent example of a program we might wish to typecheck, in order to verify that the list-abstraction transformation preserved the program's type-correctness.

The program consists of a function application, so we first typecheck the function being

```
 typecheck :: Expr -> IO (Type)
 typecheck (Var id) = return(idType id)
 typecheck (Lit (MachChar _)) = return(charTy)
 typecheck (Lit (MachInt _)) = return(intTy)
 typecheck (Con con args) = do
     let (tyargs, termargs) = span isTypeArg args
     let appliedTy = applyTys (dataConType con)
                              (map (\ (Type ty) -> ty) tyargs)
     argTys <- typecheckList termargs
     return(foldl typeApply appliedTy argTys)
 typecheck (App fun (Type tyArg)) = do
     funTy <- typecheck fun
     return (case funTy of
                f@(ForAllTy _ _) -> (applyTy f tyArg)
                _ -> error $ "typecheck: app: attempt to"
                          ++ "apply an expression of non-forall-type to a type")
-- Special case for build
 typecheck a@(App (App (Var funId) (Type someTy)) buildArg) = do
  (if (funId == buildId) then
    (typecheckBuild a)
   else
    (typecheckApp a))
 typecheck a@(App fun arg) = (typecheckApp a)
 typecheck (Lam arg body) = do
       argTy <- typecheck (Var arg)
       bodyTy <- (typecheck body)
       return(case argTy of
-- Eventually, this case should check that the type variable (<arg>) is not free
-- in the surrounding type environment, but this is a technicality.
               (TyConApp Kind _) ->  (ForAllTy arg bodyTy)
               _ -> (FunTy argTy bodyTy))
 typecheck (Let binds body) = do
   bindTys <- (typecheckBind binds)
   typecheck body
 typecheck (Case expr var alts) = do
  -- typecheck expr
     exprTy <- typecheck expr
  -- typecheck alts
     (lhsTys, rhsTys) <- typecheckAlts alts
  -- make sure all lhsTys are the same and all rhsTys are the same
     return (if (not ((allTheSame lhsTys) && (allTheSame rhsTys))) then
               error "typecheck: alternatives have different types in case"
             else
               (head rhsTys))
 typecheck (Type _) = error $ "typecheck: this shouldn't happen\n"
                             ++ "typecheck: attempt to typecheck a type"
```

Figure 3.1: The F typechecker

```
typecheckBind :: Bind -> IO([Type])
typecheckBind (NonRec var expr) = do
   varTy <- typecheck (Var var)
   exprTy <- typecheck expr
   return(if (varTy == exprTy) then [exprTy]
          else error "typecheckBind: lhs and rhs of bind don't match")
typecheckBind (Rec binds) = do
  tys <- mapM typecheckBind (map (\ (var, exp) -> (NonRec var exp)) binds)
  return(concat tys)
typecheckAlts :: [Alt] -> IO([Type], [Type])
typecheckAlts alts = (mapAndUnzipM typecheckAlt alts)
typecheckAlt :: Alt -> IO(Type, Type)
typecheckAlt (pat, vars, expr) = do
  exprTy <- (typecheck expr)
  (case pat of
-- Should really typecheck the pattern...
   (ConPat con) -> return (exprTy, exprTy)
   (LitPat lit) -> do
                   litTy <- typecheck (Lit lit)
                   return(litTy, exprTy)
   DEFAULT -> return(exprTy, exprTy))
typecheckApp (App fun arg) = do
  funTy <- typecheck fun
  argTy <- typecheck arg
  return(typeApply funTy argTy)
typeApply f@(FunTy formalTy resTy) actualTy =
 if (formalTy /= actualTy) then
    error $ "typecheck: typeApply: type mismatch applying "
           ++ (pretty f) ++ " to " ++ (pretty actualTy)
 else resTy
typeApply funty argty =
 error $ "typecheck: typeApply: attempt to apply non-function type,"
       ++ " namely " ++ (pretty funty) ++ " and " ++ (pretty argty)
typecheckBuild (App (App (Var buildId) (Type someTy)) buildArg) = do
 argTy <- typecheck buildArg
 (case argTy of
   (ForAllTy tyVar (FunTy (FunTy first (FunTy second third))
                          (FunTy fourth fifth))) ->
     (if ((first == someTy)  && (all ((==) (TyVarTy tyVar))
                                 [second, third, fourth, fifth])) then
       return(TyConApp List [someTy])
      else error $ "typecheck: argument to build has wrong type, namely "
                ++ (pretty argTy))
   _ -> error $ "typecheck: argument to build has wrong type")
typecheckList :: [Expr] -> IO([Type])
typecheckList exps = (mapM typecheck exps)
```

Figure 3.2: Auxiliary functions for the F typechecker

```
main = putStr (show (foldr (\ a r -> (a:r)) [True]))
```

Figure 3.3: A simple list-manipulating Haskell program

```
foldr : forall a. forall b. (a -> b -> b) -> b -> [a] -> b
  Bool
  [Bool]
  (\a : Bool -> \r : [Bool] ->
    Cons Bool (a : Bool) (r : [Bool]))
  (Nil Bool)
  (build : forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
     Bool
-- the * represents Kind, the type of a type
     (\resultTy : * -> \c : Bool -> resultTy -> resultTy -> \n : resultTy ->
       c : Bool -> resultTy -> resultTy True (n : resultTy)))
```

Figure 3.4: Figure 3.3, after list abstraction

applied:

```
foldr : forall a. forall b. (a -> b -> b) -> b -> [a] -> b
  Bool
  [Bool]
  (\a : Bool -> \r : [Bool] ->
    Cons Bool (a : Bool) (r : [Bool]))
  (Nil Bool)
```

This function is itself a function application, so we keep going until we get to `foldr`, which is a variable, and find that it has type ∀ `a.` ∀ `b.` `(a` → `b` → `b)` → `b` → `[a]` → `b`. Applying this type to the type arguments for `foldr`, we get the type `((Bool` → `[Bool]` → `[Bool])` → `[Bool]` → `[Bool]` → `[Bool])`. The two term arguments have types `(Bool` → `[Bool]` → `[Bool])` and `[Bool]`, so the type of the function in the application which comprises the main program is `([Bool] -> [Bool])`.

Now we check the type of the argument, which is:

```
  (build : forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
     Bool
-- the * represents Kind, the type of a type
     (\resultTy : * ->
       \c : Bool -> resultTy -> resultTy ->
        \n : resultTy ->
           c : Bool -> resultTy -> resultTy True (n : resultTy)))
```

Invoking the special rule for `build`, we first typecheck the term argument to `build`, which is:

```
(\resultTy : * -> \c : Bool -> resultTy -> resultTy -> \n : resultTy ->
       c : Bool -> resultTy -> resultTy True (n : resultTy))
```

Using the rules for type abstractions, term abstractions, and function applications, we determine that this expression has type (∀ `resultTy . (Bool` → `resultTy` → `resultTy)` → `resultTy` → `resultTy)`. This is the right form for the term argument to `build`, and in addition, `Bool` is equal to the type argument to `build` (`Bool`) and the `resultTy`s are all equal to each other, so the application of `build` has type `[Bool]`.

Now that we know the type of the function (`[Bool] -> [Bool]`), and the type of the argument (`[Bool]`), in the function application which comprises the program, the rule for function applications determines that the type of the program is `[Bool]`, which is what we would expect.

3.2 The F Interpreter

The F interpreter is somewhat less straightforward than the typechecker, mainly due to the presence of Prelude functions. For the typechecker, Prelude functions are easy to deal with, since they are represented by variable names which already have their types embedded within them. The interpreter, on the other hand, actually has to know what the functions are.

The interpreter is accessed by calling the function `eval`, which takes an F expression and returns an object of type `Result`, representing the value of the expression:

```
eval :: Expr -> IO (Result)
```

The definition of the `Result` datatype is shown in Figure 3.5.

```
data Result =
    IntRes Integer
  | CharRes Char
  | ListRes [Result]
  | BoolRes Bool
  | Closure (Var, CoreExpr, Env)
  | CurriedPrim (Primop, Int, [Result])
  | ExprRes (CoreExpr, Env)

type Primop = (Var, [Result] -> Result)
```

Figure 3.5: The `Result` datatype

A `Closure` of `v`, `expr`, and `env` represents the expression `(\ v -> expr)`, if it was defined in `env`. A `CurriedPrim` represents a Haskell Prelude function, and is represented by a triple of a `Primop`, an integer denoting the number of arguments the function takes, and a possibly empty list representing any accumulated arguments (for example, evaluating `(App (Var +) (Lit (MachInt 1)))` would result in `(CurriedPrim (`*plus*`, 2, [(IntRes 1)])`, where *plus* is the `Primop` representing the `+` function). A `Primop` is a pair of a Prelude function's name, and a function that takes a list of `Result`s, checks that they have the proper type, and if so, returns a `Result` representing the value of applying the appropriate Prelude function to the arguments' values. An `ExprRes` consists of an arbitrary F expression, along with the

environment where it was defined. The type `ExprRes` is used within the interpreter to model lazy evaluation for `Let` bindings.

The interpreter relies on two constant environments, `funenv` and `specialenv`, as well as the environment that it takes as an argument representing the variables in scope for the current expression. `funenv` maps each Prelude function's name onto either a `CurriedPrim` representing the function (with `[]` as the list of accumulated arguments), or for any higher-order functions, a `Closure` giving the function's definition in F.[1] `specialenv` contains various functions which GHC inserts into programs and are used to convert between types, such as `fromInt` (which has type `(Int -> Integer)` – `Int` and `Integer` both represent integers in Haskell, but F has only one integer type). If a variable is found in `specialenv`, it is interpreted as the identity function.

The interpreter assumes that the input expression has already been typechecked, and so it can (and does) ignore all type applications and type abstractions, treating `(\ ty -> expr)` as `expr` (if `ty` is a type variable) and `(fun ty)` as `fun` (if `ty` is a type).

The function `eval` calls the function `eval'`, which has the following type. The environment argument represents the variables which are currently in scope, mapping variable names onto `Result`s.

```
eval' :: Env -> Expr -> IO(Result)
```

The simpler cases in `eval'` are shown in Figure 3.6. For the case for `Con`, only one clause is shown; the rest of it is similar, with special cases for each constructor. The case for `Let` causes the bindings to be evaluated lazily; this was the simplest strategy, because the bindings may be mutually recursive.[2]

The case for `Var` is shown in Figure 3.7. First, the variable is looked up in the environment of variables currently in scope. If it is found, the result of the lookup is returned, unless it is an `ExprRes`, in which case the result of evaluating its expression is returned. If it is not found, it is subsequently looked up in the two environments of Prelude functions, and the result of the lookup is returned if it is found. If it still hasn't been found, an "unbound variable" error is signalled.

The case for `App` is shown in Figure 3.8. The second case checks whether the function being applied is `(build ty)`, for some `ty`. If so, it is evaluated by applying the argument to `(Lam x (Lam xs (Con cons [(Var x), (Var xs)])))` and `(Con nil [])`. This is hard-wired into the interpreter, although `build` could just as easily be included in the environment of primitive operators. If the function is not `build`, the interpreter checks whether the function is a `Closure` or a `CurriedPrim`. If the function is a `Closure`, the result of evaluating the function body in the `Closure`'s environment, extended with the function's formal parameter bound to the result of evaluating the argument, is returned. (This implements lexical scoping, as is used in Haskell.) If it is a `CurriedPrim`, the interpreter checks whether the primop

[1] Not all Prelude functions have been added as of yet, but it is easy, if tedious, to add more.

[2] Since Haskell is a lazy language, normally we would be able to handle mutually recursive bindings without any extra work. However, since the interpreter returns results in the IO monad, the only way to model laziness in a less clumsy way is to use the `fixIO` operator, which is defined in GHC's `IOExts` library. We tried to define the `Let` case for the interpreter in terms of `fixIO`, but were never able to do so without the program going into an infinite loop.

```
eval' env (Lit (MachChar c)) = return(CharRes c)
eval' env (Lit (MachInt i))  = return(IntRes i)
eval' env (Con con args) = do
   newargs <- (mapM (eval' env)
               (filter (\ x -> (not (isTypeArg x))) args))
   let len = length newargs
   (if (con == cons) then
     (if (len == 2) then
        return(listresCons (head newargs) (tail newargs))
      else
        error $ "eval: cons applied to " ++ (show len) ++
              " args, expects 2 args\n" ++ "offending expr = " ++
               (pretty c) ++ " whose args are " ++
                 (showResult (ListRes newargs)))
    else -- and so on...)
eval' env l@(Lam arg body) = do
-- check if it's a type lambda; if so, just evaluate the body
 return(case (varDetails arg) of
         TyVar -> (eval' env body)
         _ -> return(Closure(arg, body, env)))
eval' env (Let binds body) = do
 newenv <- (insertBinds (flattenBinds binds) env)
 (eval' newenv body)

insertBinds binds env = do
 return (tableExtend vars (map ExprRes exprs) env)
  where (vars, exprs) = unzip binds
```

Figure 3.6: The F evaluator – `Lit`, `Con`, `Lam`, and `Let`

```
eval' env (Var id) = do
 (case (tableLookup id env) of
  (Just (ExprRes(expr, itsenv))) -> (eval' itsenv expr)
  (Just val) -> return val
  Nothing -> (case (tableLookup id funenv) of
              (Just res) -> return res
              Nothing -> (case (tableLookupString id specialenv) of
                          (Just res) -> return res
                          Nothing -> error $ "eval: unbound variable: "
                                         ++ (pretty (idName id)))))
```

Figure 3.7: The F evaluator: `Var`

```
eval' env (App fun (Type _)) = eval' env fun
eval' env a@(App (App (Var funId) (Type _)) buildExpr) =
  if (funId == buildId) then
   (eval' env (App (App buildExpr
                         (Lam x (Lam xs
                                 (Con cons [(Var x),(Var xs)]))))
                         (Con nil [])))
  else (evalFun env a)
eval' env a@(App _ _) = evalFun env a

evalFun env (App fun (Type _)) = (eval' env fun)
evalFun env (App fun arg) =  do
  res' <- (eval' env fun)
  case res' of
    (Closure (l@(Lam v b), returnedEnv)) ->
      do
       newarg <- eval' env arg
       (eval' (tableInsert v newarg returnedEnv) b)
    (CurriedPrim (primop, numargs, args)) ->
        if (((length args) + 1) == numargs) then
           do
           arg' <- eval' env arg
           return(applyPrimop primop (args ++ [arg']))
        else
           do
           arg' <- eval' env arg
           return(CurriedPrim(primop, numargs, args ++ [arg']))
    exp -> error $ "eval: attempt to apply " ++ (showResult exp)
```

Figure 3.8: The F evaluator – `App`

application is saturated; if it is, it returns the result of applying the primop to the list of accumulated arguments with the current argument appended onto it, if it is not, it returns the same `CurriedPrim` with the current argument appended onto the list of accumulated arguments. If the function is another type of `Result`, an error is signalled.

The case for evaluating `Case` is shown in Figure 3.9. The `findMatch` function determines which alternative `(con, vars, result)` matches the scrutinee, and, like other parts of the program, relies on special cases for each possible constructor. If the scrutinee is a nonempty list, the interpreter extends the environment by mapping the members of `vars` onto the list's head and tail, and returns the result of evaluating `result` in that environment.

To see the interpreter in action, consider the program shown in Figure 3.4. The program consists of a function application, so we first evaluate the function, which gives us:

```
foldr
 Bool
 Bool
 (\a : Bool -> \r : [Bool] ->
    Cons Bool (a : Bool) (r : [Bool]))
  (Nil Bool)
```

This is itself an application, so we keep going similarly until we get to `foldr`, which is a variable. Looking up `foldr` in the environment of primitives gives us a closure whose abstraction is:[3]

```
(\ f z l -> (case l of
               (Cons x xs) -> (f x (foldr f z xs))
               Nil -> z))
```

When we apply this function to the first two arguments, we end up with a closure of the abstraction:

```
(\ l -> (case l of
          (Cons x xs) -> (f x (foldr f z xs))
          Nil -> z))
```

and an environment (call it `foldr_env`) where `f` and `z` are bound to `(\ a -> \ -> r -> Cons a r)` and `Nil`, respectively. This is the value of the operator in the application which comprises the main program.

Next we evaluate the operand in the top-level application, which is:

```
(build
    Bool
    (\resultTy -> \c -> \n ->
      c True n))
```

[3]Technically, `foldr` should have type arguments, but since the interpreter ignores type arguments, its definitions of Prelude functions do not include them.

```
eval' env (Case c var alts) = do
  caseexp <- eval' env c
  let (con, vars, result) = findMatch caseexp alts
  (eval'
    (tableInsert var caseexp
      (case caseexp of
         (ListRes [])     -> env
         (ListRes (x:xs)) ->
          (case con of
           (ConPat c) -> (if (c == cons) then
                            (tableExtend vars [x, (ListRes xs)] env)
                           else
                            env)
           _ -> error $
                  "eval': this can't happen: case pattern
                   doesn't match scrutinee")))
-- this case handles an arbitrary constructor of one argument;
-- more cases would have to be added here if other constructors
-- were added to the language
          _ -> (tableExtend vars [caseexp] env)) result)
findMatch result alts = (case result of
                         (ListRes l) -> (findMatchList l alts)
                         (IntRes i) -> (findMatchLit result alts)
                         (CharRes c) -> (findMatchLit result alts)
                         (BoolRes b) -> (findMatchCon
                                          (if b then true else false)
                                          alts)
                         (CurriedPrim (primop, _, args)) ->
                          (findMatch
                           (applyPrimop primop args)
                           alts))
findMatchList l alts = (case l of
                        [] -> (findMatchCon nil alts)
                        (x:xs) -> (findMatchCon cons alts))
findMatchCon con alts =
 (case (filter (\ x -> (case x of
                         (DEFAULT, _, _)  -> True
                         ((ConPat c), _, _) -> (conEquals c con))) alts) of
     [] -> error "findMatchCon: nonexhaustive patterns in case''
     (alt:_) -> alt)
findMatchLit litRes alts =
 (case (filter (\ thing -> case thing of
                             ((LitPat lit), _, _) ->
                                 (litEquals lit litRes)
                             (DEFAULT, _, _) -> True
                             ((ConPat con), _, _) ->
                               ((getUnique con)
                               == (getUnique isharp))) alts) of
     [] -> error "findMatchLit: nonexhaustive patterns in case"
     (alt:_) -> alt)
```

Figure 3.9: The F evaluator – `Case`

Using the special case for `build`, we apply the term argument to `(\ x xs -> (Cons x xs))` and `Nil`, so we evaluate

```
(c True n)
```

in an environment (call it `build_arg_env`) where `c` and `n` are bound to `(\ x xs -> (Cons x xs))` and `Nil`, respectively. First we evaluate `(c True)`, giving us a closure of the abstraction:

```
(\ xs -> (Cons x xs))
```

and an environment (call it `c_env`) where `x` is bound to `True`. Then we evaluate `(Cons x xs)` in `c_env` extended with a binding between `xs` and `Nil`, which gives us `(Cons True Nil)`, or `[True]`. We have now evaluated the operand in the top-level application.

Finally, we evaluate the body of the function in the top-level application, which is:

```
(case l of
  (Cons x xs) -> (f x (foldr f z xs))
  Nil -> z)
```

in `foldr_env`, extended with a binding between `l` and `[True]` (call this extended environment `foldr_env_1`). First we evaluate `l`, giving us `[True]`. The `findMatch` function tells us that `[True]` matches the case beginning with `(Cons x xs)`, so we evaluate `(f x (foldr f z xs))` in `foldr_env_1`, extended with bindings between `x` and `True`, and `xs` and `Nil`. In a similar fashion as before, this gives us `(Cons True Nil)`, or `[True]`. So the entire program evaluates to `[True]`, as expected.

Chapter 4

Shortcut Deforestation

Chitil's prototype implementation of type-inference-based deforestation only included the list abstraction algorithm, which initially produced output that contained the abstracted list constructors, `c` and `n`, as free variables. The shortcut rule itself was not implemented. After implementing the postprocessing necessary to turn an expression produced by the list abstraction algorithm into a meaningful expression (i.e., one without free variables), I implemented the shortcut rule.

4.1 The List Abstraction Algorithm

It is first necessary to understand the list abstraction algorithm itself. First, it replaces every list constructor in the input expression with a new variable, and every type of every list-valued expression with a new type variable. Then, it assigns a special type, called `build`, to the entire expression. By deriving a principal typing for the expression, it determines which type variables within it are equivalent to `build`; the list constructors that have these types are precisely the ones that can be abstracted. Finally, it replaces any constructors that were not abstracted over with their original values. The result is an expression that has had some of its list constructors replaced with variables, `c` (representing cons) and `n` (representing nil).

For example, consider the following Haskell program, which maps the `not` function onto a list of booleans and then `and`s it together:

```
main = putStr (show (foldr (\ a b -> a && b) True
                      (foldr (\a r -> (not a):r)
                       []
                       [True])))
```

The list abstraction algorithm should be applied to the inner `foldr` application, since it is immediately consumed by another `foldr`. Here is the output of the original list abstraction algorithm, applied to the translated version of that `foldr` application:

```
let { }
in
foldr : forall a. forall b. (a -> b -> b) -> b -> [a] -> b
 Bool
 build101
 (\a : Bool -> \r : build101 ->
    c4 : Bool -> build101 -> build101 (not : Bool -> Bool (a : Bool))
                                      (r : build101))
 (n3 : build101)
 (Cons-1 : forall a-10. a-10 -> [a-10] -> [a-10]
   Bool
   True-5
   (Nil-2 : forall a-10. [a-10] Bool))
```

(The Core-to-F translator expects the top-level program – a list of Core bindings, `<binds>` – to include a binding for a function called `main`, whose definition is of the form `(putStr (show <exp>`. This is an expectation that GHC makes as well. The resulting F expression is of the form `(let <binds> in <exp>)`.)

Notice that `c4` and `n3`, the abstracted list constructors, are free variables. The type variable `build101`, representing the result type of the abstract list, is also a free variable.

4.2 Postprocessing Steps

In order for an expression resulting from the list abstraction algorithm to make sense, it has to be transformed into a λ of `c` and `n`, and then have a `build` wrapped around it. I added code to do this postprocessing.

I wrote a function called `searchForFoldrs` that takes an F expression and searches for expressions of the form `(foldr ty1 ty2 f start l)`. When it finds one, it first calls itself recursively on each subexpression, in order to handle any `foldr` applications within the subexpressions. This results in new subexpressions `f'`, `start'`, and `l'`. Then, it applies the list abstraction algorithm to `l'`, resulting in an abstract list, `abstractList`. It calls another function, `insertBuild`, that turns `abstractList` into `buildExpr`, which has the form `(\ c n` $\rightarrow$ `abstractList)`. Finally, it returns the expression:

```
(foldr ty1 ty2 f' start' buildExpr)
```

The code for the relevant case of `searchForFoldrs` is shown in Figure 4.1. (The other cases merely apply `searchForFoldrs` to subexpressions.)

The function `insertBuild` takes three Uniques[1] – `nUnique`, `cUnique`, and `resultTyUnique` – a type, `elementTy`, and an F expression, `expr`. It returns the expression:

[1]A `Unique` is the type of unique identifiers for variables, and is represented as an integer. Obviously, these `Unique`s must actually be unique – that is, not used anywhere else in the program.

```
searchForFoldrs us a@(App (App (App (App (App (Var id) ty1@(Type t1))
                                       ty2@(Type t2)) f) start) arg) =
    if (nameEquals id "foldr") then
     do
      newarg <- (searchForFoldrs newus arg) -- handle subexpressions
      newf <- (searchForFoldrs us1 f)
      newstart <- (searchForFoldrs us2 start)
      case newarg of
         (Var _) -> return a
         _ -> do
-- buildListBuild is Chitil's list abstraction algorithm
          maybeBuildExpr <- (buildListBuild us' emptyUFM newarg)
          (case maybeBuildExpr of
            Just buildExpr ->
              return(App
                       (App (App (App (App (Var id) ty1) ty2) f) start)
                      -- t1 is the element type
                      (insertBuild nUnique cUnique resultTyUnique t1 buildExpr))
            Nothing -> error ("Type inference failed on expression "
                              ++ (pretty newarg)))
    else
      searchForFoldrsInApp us a
    where
-- <us> is a UniqSupply -- a data structure that can generate new
-- unique names. splitUniqSupply creates two UniqSupplies from a
-- single UniqSupply.
     (us', newus) = (splitUniqSupply us)
     (us1, us2) = (splitUniqSupply us')
     [nUnique, cUnique, resultTyUnique] = (uniqsFromSupply 3 us2)
```

Figure 4.1: searchForFoldrs

```
(build elementTy
       (\ resultTy<resultTyUnique>
          c<cUnique>
          n<nUnique> -> renamedExpr))
```

where `renamedExpr` is `expr` with `c`<`cUnique`>, `n`<`nUnique`>, and `resultTy`<`resultTyUnique`> substituted for `c4`, `n3`, and `build101`, respectively. This process of unique renaming is necessary because the list abstraction algorithm always uses the same variables, `c4` and `n3`, to represent the abstracted *cons* and *nil*, and the same type variable, `build101`, to represent the result type of an abstracted list expression.

I have finished describing what is necessary to get the result of the list abstraction algorithm into the right form for shortcut deforestation. The code for the example from above, just before shortcut deforestation is performed, is shown in Figure 4.2.

```
let { }
in
foldr : forall a. forall b. (a -> b -> b) -> b -> [a] -> b
 Bool
 Bool
 (\a : Bool -> \b : Bool -> && : Bool -> Bool -> Bool (a : Bool)
                                                      (b : Bool))
 True-5
 (build666 : forall a-10.
             (forall b-11. (a-10 -> b-11 -> b-11) -> b-11 -> b-11) -> [a-10]
   Bool
   (\resultTy3220 : * ->
     \c1610 : Bool -> resultTy3220 -> resultTy3220 ->
      \n805 : resultTy3220 -> let { } in
        foldr : forall a. forall b. (a -> b -> b) -> b -> [a] -> b
          Bool
          resultTy3220
          (\a : Bool -> \r : resultTy3220 ->
             c1610 : Bool -> resultTy3220 -> resultTy3220
              (not : Bool -> Bool (a : Bool))
              (r : resultTy3220))
          (n805 : resultTy3220)
          (build666 : forall a-10.
            (forall b-11. (a-10 -> b-11 -> b-11) -> b-11 -> b-11) -> [a-10]
            Bool
            (\resultTy6452 : * ->
              \c3226 : Bool -> resultTy6452 -> resultTy6452 ->
               \n1613 : resultTy6452 -> let { } in
                  c3226 : Bool -> resultTy6452 -> resultTy6452
                   True-5
                   (n1613 : resultTy6452)))))
```

Figure 4.2: Sample result of `searchForFoldrs`

4.3 Applying the Shortcut Rule

The shortcut rule is implemented by a function called `shortcut`. It searches for instances where `foldr` is applied to an application of `build`, of the form `(foldr ty1 ty2 f start (build buildTy abstractList))`. Then it applies `shortcut` to `f`, `start`, and `abstractList`, with results `newf`, `newstart`, and `newlist`, respectively. Finally, if `newf` has type $\alpha \rightarrow \beta \rightarrow \beta$, it returns the expression `(newlist` α `newf newstart)`. The code for `shortcut` is shown in Figure 4.3.

```
shortcut :: CoreExpr -> IO (CoreExpr)
-- foldr e_cons e_nil (build (\ c n -> <exp>))
--      => ((\ c n -> <expr>) resultTy e_cons e_nil)
-- where resultTy = type of e_nil
shortcut a@(App (App (App (App (App (Var id) ty1) ty2) f) start)
               (App (App (Var argId) buildTy) abstractList)) =
          (if ((nameEquals id "foldr") && (argId == buildId)) then
             do
              newf <- (shortcut f)
              newstart <- (shortcut start)
              newlist <- (shortcut abstractList)
              resultTy <- typecheck start
              return(App (App (App newlist (Type resultTy)) newf) newstart)
           else
              (mapExprExprM shortcut a))
shortcut expr =
     (mapExprExprM shortcut expr)
```

Figure 4.3: Code for `shortcut`

For the example above, the result of applying the shortcut rule (and β-reducing type applications afterwards) is shown in Figure 4.4.

```
let { }
  in \c1610 : Bool -> Bool -> Bool ->
      \n805 : Bool ->
        let { }
          in \c3226 : Bool -> Bool -> Bool ->
              \n1613 : Bool -> let { }
                in c3226 : Bool -> Bool -> Bool True-5 (n1613 : Bool)
             (\a : Bool -> \r : Bool ->
                c1610 : Bool -> Bool -> Bool (not : Bool -> Bool (a : Bool))
                                             (r : Bool))
             (n805 : Bool)
       (\a : Bool -> \b : Bool ->
         && : Bool -> Bool -> Bool (a : Bool)
                                  (b : Bool))
       True-5
-- This beta-reduces to:
-- && (not True) (True)
```

Figure 4.4: Sample result of the shortcut rule

Chapter 5

Conclusions and Future Work

My work has advanced the type inference algorithm to the point where, with some minor improvements, it can be run on Haskell benchmarks and its performance can be compared to that of other deforestation algorithms. Unlike before, it performs the entire process of deforestation, and it can take the text of a Haskell program and output the the text of a Haskell program, as opposed to before, when it took an F AST (abstract syntax tree) and returned an F AST.

5.1 Working Examples

I have tested the implementation of type-inference-based deforestation on a number of simple Haskell programs. These programs share several properties, which reflect limitations of the current implementation:

1. The only datatypes they use are `List`, `Bool`, `Char`, and `Int`.

2. The programs that manipulate integers contain explicit type declarations, in order to avoid introducing type classes (which aren't handled by the type inference algorithm.)

3. They only use a limited range of Prelude functions (those which have been hardcoded into the interpreter).

4. All functions which are defined in terms of `foldr` have been cloned by hand.

5. They all consist of simple list-manipulating functions, which are defined in terms of `foldr`.

The first limitation would have to be addressed by extending the syntax of F to include more type constructors and data constructors. It would be fairly simple to extend the range of basic types to those found in Core (i.e., adding strings, floats, tuples, and so forth), but adding user-defined datatypes would require some thought. Some minor changes to the code for the type inference algorithm itself would be necessary, for example, the function that determines the type of an expression would need to be extended with more cases for the new types. Changes to the Core-to-F translator, specifically the part of it that translates types,

would also be necessary, in order to detect specific type constructors in Core for the new types and translate them into the right F type constructors. Finally, the F typechecker and interpreter would have to be extended to handle the new types. To summarize, it would be desirable to modify all the code to handle data constructors in a more general way than it currently does.

Removing the second limitation would only require modifying the Core-to-F translator to detect type class types and translate them into something simpler. For example, if you define a function that operates on integers and don't give an explicit type for it, GHC will infer a type for it that is defined in terms of a type class like `(Num a)`. The translator could check for such types and translate them into one of the types that F supports – `Int`, for example.

The third limitation could be addressed simply by adding more Prelude functions into the F interpreter. Of course, it is already possible to run type-inference-based deforestation on programs which use Prelude functions which are not yet defined in the interpreter – it just isn't possible to test the results using the interpreter.

The other two limitations are addressed below.

5.2 Future Work

Here are some future directions in which my work could be taken:

5.2.1 Type-Based Cloning

Along with the type-inference-based deforestation algorithm, Chitil described a type-based method for cloning recursive and non-recursive function definitions, known as the worker/wrapper scheme [Chi00]. For the type inference algorithm to run on ordinary programs, it will be necessary to implement the worker/wrapper scheme – otherwise, cloning must be done by hand.

For example, the following program, which computes the sum of squares of the list `[10..1]`:

```
main = putStr (show (sum (map (\x -> (x * x)) [10..1])))
```

has to be manually transformed into the following form in order for the type inference algorithm to work:

```
main = putStr (show (foldr (\ a b -> a + b) 0
                    (foldr (\a r -> (a*a) : r)
                     []
                     (let mydown :: Int -> [Int]
                          mydown x =
                            (if (x == 0) then
                              []
                             else
                              (x : (mydown (x - 1))))
                      in
                      (mydown 10)))))
```

As is clear from this example, this process is extremely tedious and makes it impractical to test the current implementation on any programs longer than a few lines.

I believe that automatic cloning is the only major addition that would be necessary in order for the current implementation to deforest any programs which manipulate lists in a uniform way.

5.2.2 Beyond Lists

Chitil's description of the deforestation algorithm only handles lists, but the algorithm could be extended to datatypes besides lists – i.e., trees. Lists are the simplest and most commonly used data structure in functional programs, but other datatypes could be amenable to deforestation as well. For example, if deforestation were extended to eliminate the datatype of abstract syntax trees, a compiler that does multiple optimization passes could be automatically fused into a single pass. This would require modifying the implementation so that it can generalize the notion of `foldr` to arbitrary datatypes (the theory behind this idea is described in [MFP91]).

5.2.3 Benchmarking

If the above changes are made, it would then be possible to test type-inference-based deforestation on a wide range of benchmarks, such as those in the `nofib` suite for Haskell. Since many of these benchmarks are multi-module programs, and the type inference algorithm only handles single-module programs, this assumes the existence of a demodulizer to convert multi-module programs into single-module programs.

5.2.4 Integration with GHC

If type-inference-based deforestation becomes practical to apply to arbitrary programs, it would be desirable to integrate type-inference-based deforestation into GHC, as an optimization pass. This would require finding a way to translate from F back into Core, retaining all the information that was stored into the original Core AST. An alternative strategy would be to modify the implementation to operate on Core programs instead of F programs, but

this would require some fairly major changes to the list abstraction algorithm and supporting code.

To give a sense for the amount of information that the Core-to-F translator discards, consider the datatype definition for GHC's `Var`:

```
data Var
  = Var  {
        varName    :: Name,
        realUnique :: Int#,
        varType    :: Type,
        varDetails :: VarDetails,
        varInfo    :: IdInfo
      }
```

and the type definition for just *one* field of this record type:

```
data IdInfo
  = IdInfo  {
        flavourInfo     :: IdFlavour,
        arityInfo       :: ArityInfo,
        demandInfo      :: Demand,
        specInfo        :: CoreRules,
        strictnessInfo  :: StrictnessInfo,
        workerInfo      :: WorkerInfo,
        unfoldingInfo   :: Unfolding,
        updateInfo      :: UpdateInfo,
        cafInfo         :: CafInfo,
        cprInfo         :: CprInfo,
        lbvarInfo       :: LBVarInfo,
        inlinePragInfo  :: InlinePragInfo,
        occInfo         :: OccInfo
      }
```

It should be clear from the complexity of these types that saving and restoring all the information they contain would be an extremely nontrivial task. But it would be a necessary task in order to integrate type-inference-based deforestation into GHC as an optimization pass. This in turn would be necessary in order to compare type-inference-based deforestation to other deforestation methods which have been implemented in the past (such as warm fusion and shortcut deforestation), because the most accurate picture of how an optimization works in practice can only be obtained by understanding how it interacts with other optimizations, such as those performed by GHC.

Bibliography

[Chi99] Chitil, Olaf, “Type Inference Builds a Short Cut to Deforestation”, Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP ’99), pp. 249-260, 1999.

[Chi00] Chitil, Olaf, “Type-Inference Based Deforestation of Functional Programs”, Ph.D thesis, Aachen University of Technology, 2000

[Gil96] Gill, Andrew, “Cheap deforestation for Non-strict Functional Languages”, Ph.D thesis, Glasgow University, 1996

[GLP93] Gill, Andrew, John Launchbury, and Simon Peyton Jones, “A Short Cut to Deforestation”, Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA ’93), pp. 223-232, 1993.

[Hug89] Hughes, John, “Why Functional Programming Matters”, *Computer Journal* v. 32 n. 2, pp. 98-107, 1989.

[LS95] Launchbury, John, and Tim Sheard, “Warm Fusion: Deriving Build-Catas from Recursive Definitions”, Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA ’95), pp. 314-322, 1995.

[MFP91] Meijer, Erik, Maarten Fokkinga, and Ross Paterson, “Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire”. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, LNCS 523, pp. 124-144. Springer, June 1991

[Ném00] Németh, László, “Catamorphism Based Program Transformations for Non-Strict Functional Languages”, Ph.D thesis, University of Glasgow, 2000

[Wad90] Wadler, Philip, “Deforestation: transforming programs to eliminate trees”, *Theoretical Computer Science*, v. 73, pp. 231-248, 1990.

Appendix A

Haskell Code

A.1 List of Modules

`Main.hs` is the main driver for the system; it uses code from GHC, `ParseFile.hs`, and `TransCore.hs` to translate the Haskell input into F, then invokes `RunTypeInference.hs` to do the rest of the work.

`Typecheck.hs` is the F typechecker and `Eval.hs` is the F interpreter. `Eval.hs` uses auxiliary code in `Table.hs` and `Result.hs`. `F2Haskell.hs` is the F-to-Haskell translator.

The previous files were all primarily written by me. The remaining files, which implement the syntax of F and the list abstraction algorithm, were primarily written by Olaf Chitil (except for `CountIO.hs`, which was added by me), with changes by me where indicated.

A.2 Main.hs

```
-- Driver that calls GHC functions, performs Core-to-F translation,
-- and then hands control to driver for type inference
-- By Kirsten Chevalier, with code from GHC's Main.hs
module Main where
import qualified Outputable
import qualified Desugar
import qualified TcModule
import qualified RnMonad
import qualified Rename
import qualified UniqSupply
import qualified Module
import qualified ParseFile
import qualified Core
import TransCore
import PrettyCore
import CoreSyn
import Var
import Name(mkSysLocalName, nameOccName)
import OccName(decode, occNameString)
import Unique
import FastString
import TypeRep
```

```
import TysWiredIn
import IdInfo
import Literal
import ErrUtils
import PprCore
import TysPrim
import RunTypeInference
import System
import List
import Eval(eval)
import Result

test :: Core.CoreExpr -> IO (Bool)
test expr = testTypeInference [] expr

main = do
--------------- copied from GHC's Main.lhs, and trimmed -----------------
  { -- get filename from command line:
    (infile:_) <- getArgs;
    -- parse the input file:
    (mod_name, rdr_module) <- ParseFile.parseFile infile;
    rn_uniqs <-  UniqSupply.mkSplitUniqSupply 'r'; -- renamer
    tc_uniqs <-  UniqSupply.mkSplitUniqSupply 'a'; -- typechecker
    ds_uniqs <-  UniqSupply.mkSplitUniqSupply 'd'; -- desugarer
    maybe_rn_stuff <- Rename.renameModule rn_uniqs rdr_module;
    (desugared, _, _, _, _) <-
--------------------------  Renaming  ----------------
    case maybe_rn_stuff of
    {
     Just (this_mod, rn_mod, iface_file_stuff, rn_name_supply, _) ->
--------------------------  Typechecking ----------------

    TcModule.typecheckModule tc_uniqs rn_name_supply
    iface_file_stuff rn_mod >>= \ maybe_tc_stuff ->
case maybe_tc_stuff of {
   Just tc_results  ->
--------------------------  Desugaring ----------------

   Desugar.deSugar this_mod ds_uniqs tc_results}};
    -- We assume there's a function called main, and use that as the body
    -- of a let expression, with the rest of the bindings as the decls.

    -- <res> indicates whether the original expression and the deforested
    -- expression evaluated to the same value
    res <- test (toFLet desugared);
    putStrLn (show res)
    }
--------------end code copied from GHC's Main.lhs -----------------

-- assumes: there is a function main = putStr (show foo)
-- foo becomes the body of the let
toFLet :: [CoreBind] -> Core.CoreExpr
toFLet binds =
```

```
 (Core.Let (transFlattenedBinds (removeGarbage flat))
   (case (pluck(findName "main" flat)) of
    (_, (App _ (App _ bdy))) -> (trans bdy)
    _ -> error "main is not of the form <putStr (show foo)>"))
   where
   flat = (CoreSyn.flattenBinds binds)
   findName :: String -> [(Var, CoreExpr)] -> Maybe(Var, CoreExpr)
   findName str binds =
   -- returns the binding for the variable <str>
       (find (\ (bndr, _) ->
      (((decode
 (occNameString
  (nameOccName
   (varName bndr)))) == str)))
binds)
   removeGarbage :: [(Var, CoreExpr)] -> [(Var, CoreExpr)]
   removeGarbage binds =
   -- removes bindings for main, show, $dshow
       (filter (\ (bndr, _) -> let thisName = (decode
       (occNameString
(nameOccName
 (varName bndr))))
in
                ((thisName /= "main") &&
 (thisName /= "show") && (thisName /= "$dShow"))) binds)
   mkApp :: Core.CoreExpr -> [Core.CoreExpr] -> Core.CoreExpr
   mkApp f args = foldl Core.App f args

pluck :: Maybe a -> a
pluck (Just x) = x
pluck Nothing = error "Main: pluck invoked on Nothing"
```

A.3 RunTypeInference.hs

```
-- Supplies the testTypeInference function, which takes an F AST,
-- applies list abstraction and shortcut deforestation to it, and
-- typechecks and interprets it
-- By Kirsten Chevalier
module RunTypeInference where

import Prelude
import List
import Core
import PrettyCore
import TypeInference
import MonadTransformer
import ThisUnique
import Typecheck
import ThisUtil
import Eval
```

```
import Result
import F2Haskell
import Build
import Maybe

testTypeInference :: [(Id,CoreExpr)] -> CoreExpr -> IO (Bool)

testTypeInference binds expr = do
  -- putStrLn "\nInput Expression:";
  -- prettyIO expr;
  --putStrLn $ "buildListBuild";
  --firsttest <- (buildListBuild (mkMyUniqSupply 100) emptyUFM expr);
  --prettyIO (fromJust firsttest);
  exprTy <- typecheck expr;
  exprVal <- eval expr;
  abstracted <- (searchForFoldrs (mkMyUniqSupply 100) expr);
  putStrLn $ "Performing list abstraction"
  --prettyIO abstracted;
  abstractedTy <- typecheck abstracted;
  abstractedVal <- eval abstracted;
  deforested <- (shortcut abstracted);
  putStrLn $ "Applying shortcut rule";
  --prettyIO deforested;
  deforestedTy <- typecheck deforested;
  deforestedVal <- eval deforested;
  -- This doesn't simplify all type applications.
  -- Haven't checked yet whether f2haskell works after this.
  simplified <- simplify deforested
  putStrLn $ "Simplifying type applications"
  prettyIO simplified
  simplifiedTy <- typecheck simplified;
  simplifiedVal <- eval simplified;

  let typecheckWorked = (allTheSame [exprTy, abstractedTy, deforestedTy, simplifiedTy]);
  let evalWorked = (allTheSame [exprVal, abstractedVal, deforestedVal, simplifiedVal]);

  (if typecheckWorked then
     putStrLn $ "Type checking succeeded! All expressions have type " ++ (pretty exprTy)
   else
     putStrLn $ "Type mismatch: \n Input expression has type "
              ++ (pretty exprTy) ++ "\nList-abstracted expression has type "
              ++ (pretty abstractedTy) ++ "\nDeforested expression has type "
              ++ (pretty deforestedTy) ++ "\nSimplified expression has type "
              ++ (pretty simplifiedTy))
  (if evalWorked then
     putStrLn $ "Evaluation succeeded! All expressions have value " ++ (show exprVal)
   else
     putStrLn $ "Value mismatch: \n Input expression has value "
              ++ (show exprVal) ++ "\nList-abstracted expression has value "
              ++ (show abstractedVal) ++ "\nDeforested expression has value "
              ++ (show deforestedVal) ++ "\nSimplified expression has value "
              ++ (show simplifiedVal))
  return(typecheckWorked && evalWorked)
```

```
shortcut :: CoreExpr -> IO (CoreExpr)
-- foldr e_cons e_nil (build (\ c n -> <exp>))
--      => ((\ c n -> <expr>) resultTy e_cons e_nil)
-- where resultTy = type of e_nil
shortcut a@(App (App (App (App (App (Var id) ty1) ty2) f) start)
       (App (App (Var argId) buildTy) abstractList)) =
  (if ((nameEquals id "foldr") && (argId == buildId)) then
     do
              newf <- (shortcut f)
              newstart <- (shortcut start)
              newlist <- (shortcut abstractList)
              resultTy <- typecheck start
              let res = (App (App (App newlist (Type resultTy)) newf) newstart)
              return res
           else
              (mapExprExprM shortcut a))
shortcut expr =
     (mapExprExprM shortcut expr)

insertBuild ::  Unique -> Unique -> Unique -> Type -> CoreExpr -> CoreExpr
insertBuild newNilPlaceholderUnique newConsPlaceholderUnique resultTyUnique
            elementTy expr =
-- Renames c4 and n3 to c<newConsPlaceholderUnique> and n<newNilPlaceholderUnique>
-- Renames build101 to resultTy<resultTyUnique>
-- Binds these three variables with a a lambda
-- Applies (build elementTy) to the resulting lambda abstraction
  (App
   (App (Var buildId) (Type elementTy))
   (Lam resultTy (Lam consPlaceholder
                (Lam nilPlaceholder
   (replaceBuild (replace (listToUFM [(nilPlaceholderUnique, nilPlaceholder),
                              (consPlaceholderUnique, consPlaceholder)])
                                  expr))))))
  where
  -- replace build101 with resultTy
   replaceBuild expr = (substExpr (listToUFM [(buildTyVar, resultTyVar)]) expr)
   replace newUniques (Var id) = (Var (replaceId newUniques id))
   replace newUniques expr = (mapExprExpr (replace newUniques) expr)
   replaceId newUniques id =
      (case (lookupUFM newUniques (idUnique id)) of
         (Just newPlaceholder) -> newPlaceholder
         Nothing -> id)
   -----------------------------------
   nilTy = resultTyVar
   consTy =  (FunTy elementTy (FunTy resultTyVar resultTyVar))
   nilPlaceholder = (mkSysLocal "n" newNilPlaceholderUnique nilTy)
   consPlaceholder = (mkSysLocal "c" newConsPlaceholderUnique consTy)
   resultTy = (smpTyVar "resultTy" resultTyUnique)
   resultTyVar = (TyVarTy resultTy)

searchForFoldrs :: UniqSupply -> CoreExpr -> IO (CoreExpr)
--    searches for expressions of the form <foldr f start l>
```

```
--    and applies buildListBuild to l
searchForFoldrs us (v@(Var _)) = return v
searchForFoldrs us (l@(Lit _)) = return l
searchForFoldrs us c@(Con con []) = return c
searchForFoldrs us (Con con (arg:args)) =
     do
     newarg <- searchForFoldrs us1 arg
     rest <- (searchForFoldrs us2 (Con con args))
     case rest of
       (Con con args') -> return (Con con (newarg:args'))
     where
     (us1, us2) = (splitUniqSupply us)
searchForFoldrs us a@(App (App (App (App (App (Var id) ty1@(Type t1))
                                       ty2@(Type t2)) f) start) arg) =
    if (nameEquals id "foldr") then
     do
      newarg <- (searchForFoldrs newus arg) -- handle subexpressions
      newf <- (searchForFoldrs us1 f)
      newstart <- (searchForFoldrs us2 start)
      case newarg of
-- if arg is just a variable, bail out. really should pass in bindings instead...
-- This won't be necessary once cloning is implemented.
 (Var _) -> return a
         _ ->
              do
      maybeBuildExpr <- (buildListBuild us' emptyUFM newarg)
      (case maybeBuildExpr of
       Just buildExpr -> return(App (App (App (App (App (Var id) ty1) ty2) f) start)
-- t1 is the element type
        (insertBuild nUnique cUnique resultTyUnique t1 buildExpr))
       Nothing -> error ("Type inference failed on expression " ++ (pretty newarg)))
    else
      searchForFoldrsInApp us a
    where
     (us', newus) = (splitUniqSupply us)
     (us1, us2) = (splitUniqSupply us')
     [nUnique, cUnique, resultTyUnique] = (uniqsFromSupply 3 us2)
searchForFoldrs us a@(App fun arg) = (searchForFoldrsInApp us a)
searchForFoldrs us (Lam arg body) =
     do
     body' <- (searchForFoldrs us body)
     return (Lam arg body')
searchForFoldrs us (Let binds body) =
    do
    binds' <- (searchForFoldrsInBinds us1 binds)
    body' <- (searchForFoldrs us2 body)
    return (Let binds' body')
    where
    (us1, us2) = splitUniqSupply us
searchForFoldrs us (Case exp var alts) =
    do
    exp' <- (searchForFoldrs us1 exp)
    alts' <- (searchForFoldrsInAlts us2 alts)
```

```
    return (Case exp' var alts')
    where
    (us1, us2) = splitUniqSupply us
searchForFoldrs us (t@(Type ty)) = return t

searchForFoldrsInApp us (App fun arg) =  do
    fun' <- (searchForFoldrs us1 fun)
    arg' <- (searchForFoldrs us2 arg)
    return (App fun' arg')
    where
    (us1, us2) = splitUniqSupply us
searchForFoldrsInBinds :: UniqSupply -> CoreBind -> IO (CoreBind)
searchForFoldrsInBinds us (NonRec var exp) =
       do
       exp' <- (searchForFoldrs us exp)
       return (NonRec var exp')
searchForFoldrsInBinds us r@(Rec []) = return r
searchForFoldrsInBinds us (Rec ((var, exp):binds)) =
      do
      newexp <- searchForFoldrs us1 exp
      rest <- searchForFoldrsInBinds us2 (Rec binds)
      case rest of
(Rec newbinds) -> return (Rec ((var, newexp):newbinds))
      where
      (us1, us2) = (splitUniqSupply us)
searchForFoldrsInAlts :: UniqSupply -> [Alt Var] -> IO ([Alt Var])
searchForFoldrsInAlts us [] = return []
searchForFoldrsInAlts us ((pat, var, exp):alts) =
      do
      newexp <- searchForFoldrs us1 exp
      rest <- searchForFoldrsInAlts us2 alts
      return ((pat, var, newexp):rest)
      where
      (us1, us2) = (splitUniqSupply us)

simplify :: CoreExpr -> IO (CoreExpr)
-- beta-reduces any type applications in exp
simplify (App l@(Lam _ _) t@(Type _)) =
  return (tyBetaReduce l [t])
simplify expr = mapExprExprM simplify expr
```

A.4 ParseFile.hs

```
-- * ParseFile contains a bit of code stolen from
```

```
-- ghc-4.08/fptools/ghc/compiler/main/Main.lhs
-- By Kate Golder

module ParseFile where
import IO ( hPutStr, stderr )
import HsSyn
import BasicTypes ( NewOrData(..) )

import RdrHsSyn ( RdrNameHsModule )
import FastString ( mkFastCharString, unpackFS )
import StringBuffer ( hGetStringBuffer )
import Parser ( parse )
import Lex ( PState(..), P, ParseResult(..) )
import SrcLoc ( mkSrcLoc )

import Rename ( renameModule )
import RnMonad ( InterfaceDetails(..) )

import MkIface ( startIface, ifaceDecls, endIface )
import TcModule ( TcResults(..), typecheckModule )
import Module ( ModuleName, moduleNameUserString )
import CmdLineOpts
import ErrUtils ( ghcExit )
import Outputable

parseFile :: String -> IO (ModuleName, RdrNameHsModule)
parseFile filename = do
    {
     buf <- hGetStringBuffer True filename;
      let parsed = (parse buf PState{ bol = 0#,
    atbol = 1#,
              context = [],
    glasgow_exts = glaexts,
            loc = mkSrcLoc src_filename 1 })
         in do{
            case parsed of
PFailed err -> do
                                       putStrLn "parse error"
       printErrs err
       ghcExit 1
       return (error "parseModule")
POk _ m@(HsModule mod _ _ _ _ _ _) -> return (mod, m)}
 }
   where
 glaexts | opt_GlasgowExts = 1#
 | otherwise       = 0#
```

A.5 TransCore.hs

```
-- Core to F translation
-- By Kirsten Chevalier
module TransCore
where
import qualified Core
import qualified CoreSyn
import qualified IdInfo
import qualified DataCon
import qualified Literal
import qualified Name
import qualified OccName
import qualified Unique
import qualified Var
import qualified TypeRep
import qualified TysWiredIn
import qualified TysPrim
import qualified Id
import PrelBase
import PprCore
import PprType
import PrettyCore
import Outputable
import GlaExts -- necessary in order to use S#

-- takes a Core expression and returns the corresponding F expression
trans :: CoreSyn.CoreExpr -> Core.CoreExpr
trans (CoreSyn.Var id) =
    case (Id.idFlavour id) of
      (IdInfo.DataConWrapId con) ->
  (if (con == TysWiredIn.trueDataCon) then
   (Core.Con Core.true [])
   else if (con == TysWiredIn.falseDataCon) then
   (Core.Con Core.false [])
   else
   (Core.Var (transId id)))
      (IdInfo.DataConId con) ->
  (if (con == TysWiredIn.trueDataCon) then
   (Core.Con Core.true [])
   else if (con == TysWiredIn.falseDataCon) then
   (Core.Con Core.false [])
   else
   (Core.Var (transId id)))
      _ -> (Core.Var (transId id))
trans (CoreSyn.Lit l)             = (Core.Lit (transLit l))
-- Cons
trans a@(CoreSyn.App
       (CoreSyn.App (CoreSyn.App (CoreSyn.Var var) ty1) arg1) arg2) =
```

```
    case (Id.idFlavour var) of
     (IdInfo.DataConId con)     -> handleCons con a
     (IdInfo.DataConWrapId con) -> handleCons con a
     otherwise                  -> (transApp a)
  where handleCons con a@(CoreSyn.App
  (CoreSyn.App
   (CoreSyn.App
    (CoreSyn.Var var)
    ty1)
   arg1)
  arg2) = if (con == TysWiredIn.consDataCon) then
                 (Core.Con Core.cons [(trans ty1),
(trans arg1),
(trans arg2)])
 else
 (transApp a)
-- Nil
trans a@(CoreSyn.App (CoreSyn.Var var) ty1) =
    case (Id.idFlavour var) of
     (IdInfo.DataConId con)      -> handleNil con a
     (IdInfo.DataConWrapId con)  -> handleNil con a
     otherwise                   -> (transApp a)
 where handleNil con a@(CoreSyn.App
(CoreSyn.Var var)
ty1) = if (con == TysWiredIn.nilDataCon) then
                                (Core.Con Core.nil [(trans ty1)])
      else
       (transApp a)
trans a@(CoreSyn.App _ _)         = (transApp a)
trans (CoreSyn.Lam arg exp)       = (Core.Lam (transId arg) (trans exp))
trans (CoreSyn.Let binds exp)     = (Core.Let (transBinds binds) (trans exp))
trans (CoreSyn.Case exp var alts) = (Core.Case (trans exp)
     (transId var)
     (transAlts alts))
trans (CoreSyn.Note _ exp)        = (trans exp)
trans (CoreSyn.Type ty)           = (Core.Type (transTy ty))

transApp :: CoreSyn.CoreExpr -> Core.CoreExpr
transApp (CoreSyn.App
  fun@(CoreSyn.Var id)
  arg) =  case (IdInfo.flavourInfo (Var.varInfo id)) of
                    IdInfo.DataConId con -> (transCon id arg)
                    otherwise            -> (Core.App (trans fun) (trans arg))
transApp (CoreSyn.App fun arg) = (Core.App (trans fun) (trans arg))

transBinds :: CoreSyn.CoreBind -> Core.CoreBind
transBinds (CoreSyn.NonRec var exp) = (Core.NonRec (transId var) (trans exp))
transBinds (CoreSyn.Rec binds)      = (Core.Rec (zip (map transId names)
 (map trans exps)))
                                       where (names, exps) = (unzip binds)
transFlattenedBinds :: [(CoreSyn.CoreBndr, CoreSyn.CoreExpr)] -> Core.CoreBind
transFlattenedBinds binds = (transBinds (CoreSyn.Rec binds))
```

```
transAlts :: [CoreSyn.CoreAlt] -> [Core.CoreAlt]
transAlts alts = (map transAlt alts)
transAlt :: CoreSyn.CoreAlt -> Core.CoreAlt
transAlt (altcon, args, exp) =
    (case altcon of
     CoreSyn.DataAlt dc -> (if (dc == TysWiredIn.nilDataCon) then
                            (Core.ConPat (Core.nil))
                            else (if (dc == TysWiredIn.consDataCon) then
                                  (Core.ConPat (Core.cons))
                                  else (if (dc == TysWiredIn.trueDataCon) then
                                         (Core.ConPat (Core.true))
                                        else (if (dc == TysWiredIn.falseDataCon) then
                                               (Core.ConPat (Core.false))
                                              else
                                                (Core.ConPat
                 (Core.C (transName
   (DataCon.dataConName dc))))))))
     CoreSyn.LitAlt  l  -> Core.LitPat (transLit l)
     CoreSyn.DEFAULT    -> Core.DEFAULT,
     (map transId args), (trans exp))

transName :: Name.Name -> Core.Name
transName nm = Core.mkSysLocalName (S# (Unique.u2i (Name.nameUnique nm)))
                (OccName.decode (OccName.occNameString (Name.nameOccName nm)))

transLit :: Literal.Literal -> Core.Literal
transLit l       = case l of
                   Literal.MachChar c -> Core.MachChar c
                   Literal.MachInt i  -> Core.MachInt i
    otherwise          -> error ("TransCore: type of " ++
 (show l) ++
 " not supported")
transId :: Var.Id -> Core.Id
transId v = (Core.mkIdVar (transName (Var.idName v)) (transTy (Var.idType v)))

transTy :: TypeRep.Type -> Core.Type
transTy (TypeRep.TyVarTy tv) = (Core.TyVarTy (transId tv))
transTy (TypeRep.AppTy t1 t2) = error "transTy: AppTy"
transTy (t@(TypeRep.TyConApp tc tys)) =
    (Core.TyConApp  (if (tc == TysWiredIn.listTyCon) then
     Core.List
     else if (tc == TysWiredIn.pairTyCon) then
     Core.Tuple
     else if (tc == TysWiredIn.boolTyCon) then
     Core.Bool
     else if (tc == TysWiredIn.intTyCon) then
     Core.Int
     else if (tc == TysWiredIn.charTyCon) then
     Core.Char
     else if (tc == TysPrim.intPrimTyCon) then
     Core.Int
                    else if (tc == TysPrim.charPrimTyCon) then
     Core.Char
```

```
     else
     Core.Int) -- iffy default case
     (map transTy tys))
transTy (TypeRep.FunTy t1 t2) = (Core.FunTy (transTy t1) (transTy t2))
transTy (TypeRep.NoteTy tn ty) = transTy ty
transTy (TypeRep.ForAllTy tv ty) = (Core.ForAllTy (transId tv) (transTy ty))

transCon :: Var.Var -> CoreSyn.CoreExpr -> Core.CoreExpr
transCon con arg = (Core.App (Core.Var (transId con)) (trans arg))
```

A.6 Typecheck.hs

```
-- A type checker for F
-- By Kirsten Chevalier

module Typecheck
 where
 import Core
 import PrettyCore
 import TypeInference
 import ThisUnique
 import List
 import ThisUtil
 import Monad
 import Build

-- For debugging
 say :: CoreExpr -> IO()
 say exp = return()
--
--putStrLn $ "typechecking " ++  (pretty exp)

 sayR :: CoreExpr -> Type -> IO()
 sayR exp ty = return()
--
--putStr ("typecheck: " ++ (pretty exp) ++ " has type " ++ (pretty ty) ++ "\n")

 typecheck :: CoreExpr -> IO (Type)

 typecheck v@(Var id) = do
     say v
     let res = idType id
     sayR v res
     return res

 typecheck l@(Lit (MachChar _)) = do
```

```
    say l
    let res = charTy
    sayR l res
    return res
 typecheck l@(Lit (MachInt _)) = do
    say l
    let res = intTy
    sayR l res
    return intTy

 typecheck c@(Con con args) = do
     say c
     let (tyargs, termargs) = span isTypeArg args
     let appliedTy = applyTys (dataConType con)
                              (map (\ (Type ty) -> ty) tyargs)
     argTys <- typecheckList termargs
     let res = (foldl typeApply appliedTy argTys)
     sayR c res
     return(res)

 typecheck a@(App fun (Type tyArg)) = do
     say a
     funTy <- typecheck fun
     let res = (case funTy of
                f@(ForAllTy _ _) -> (applyTy f tyArg)
                _ -> error $ "typecheck: app: attempt to"
                             ++ "apply an expression of non-forall-type to a type")
     sayR a res
     return res

-- Special case for build
 typecheck a@(App (App (Var funId) (Type someTy)) buildArg) = do
  (if (funId == buildId) then
    (typecheckBuild a)
   else
    (typecheckApp a))

 typecheck a@(App fun arg) = (typecheckApp a)

 typecheck l@(Lam arg body) = do
       say l
       argTy <- typecheck (Var arg)
       bodyTy <- (typecheck body)
       let res = (case argTy of
-- Eventually, this case should check that the type variable (<arg>) is not free
-- in the surrounding type environment, but this is a technicality.
           (TyConApp Kind _) ->  (ForAllTy arg bodyTy)
           _ -> (FunTy argTy bodyTy))
       sayR l res
       return res

 typecheck l@(Let binds body) = do
   say l
```

```
   bindTys <- (typecheckBind binds)
   res <- typecheck body
   sayR l res
   return res

 typecheck c@(Case expr var alts) = do
     say c
  -- typecheck expr
     exprTy <- typecheck expr
  -- typecheck alts
     (lhsTys, rhsTys) <- typecheckAlts alts
  -- make sure all lhsTys are the same and all rhsTys are the same
     let res = (if (not ((allTheSame lhsTys) && (allTheSame rhsTys))) then
                 error "typecheck: alternatives have different types in case"
                else
                 (head rhsTys))
     sayR c res
     return res

 typecheck (Type _) = error $ "typecheck: this shouldn't happen\n"
             ++ "typecheck: attempt to typecheck a type"

 typecheckBind :: CoreBind -> IO([Type])
 typecheckBind (NonRec var expr) = do
    varTy <- typecheck (Var var)
    exprTy <- typecheck expr
    return(if (varTy == exprTy) then
             [exprTy]
           else
             error "typecheckBind: lhs and rhs of bind don't match")
 typecheckBind (Rec binds) = do
   tys <- mapM typecheckBind (map (\ (var, exp) -> (NonRec var exp)) binds)
   return(concat tys)

 typecheckAlts :: [CoreAlt] -> IO([Type], [Type])
 typecheckAlts alts = do
  firstTy <- (typecheck (thrd3 (head alts)))
  (mapAndUnzipM typecheckAlt alts)
 typecheckAlt :: CoreAlt -> IO(Type, Type)
 typecheckAlt (pat, vars, expr) = do
 -- typecheck expr
 -- check that pat matches vars
   exprTy <- (typecheck expr)
   (case pat of
-- This should really reconstruct the type arguments for the
-- constructor and typecheck the constructor application corresponding
-- to the pattern.
-- should check the LHS, but for now assume it has same type as RHS
    (ConPat con) -> return (exprTy, exprTy)
    (LitPat lit) -> do
                    litTy <- typecheck (Lit lit)
                    return(litTy, exprTy)
    DEFAULT -> return(exprTy, exprTy))
```

```
 typecheckApp a@(App fun arg) = do
   say a
   funTy <- typecheck fun
   argTy <- typecheck arg
   let res = typeApply funTy argTy
   sayR a res
   return res

 typecheckBuild a@(App (App (Var buildId) (Type someTy)) buildArg) = do
  argTy <- typecheck buildArg
  (case argTy of
-- Using == is *not* sketchy here!
    (ForAllTy tyVar (FunTy (FunTy first (FunTy second third))
                           (FunTy fourth fifth))) ->
      (if ((first == someTy)
          && (all ((==) (TyVarTy tyVar))
             [second, third, fourth, fifth])) then
        return(TyConApp List [someTy])
       else
       error $ "typecheck: argument to build has wrong type, namely " ++ (pretty argTy))
    _ -> error $ "typecheck: argument to build has wrong type")

 typeApply f@(FunTy formalTy resTy) actualTy =
  if (formalTy /= actualTy) then
     error $ "typecheck: typeApply: type mismatch applying "
             ++ (pretty f) ++ " to " ++ (pretty actualTy)
  else
     resTy

 typeApply funty argty =
  error $ "typecheck: typeApply: attempt to apply non-function type,"
        ++ " namely " ++ (pretty funty) ++ " and " ++ (pretty argty)

 typecheckList :: [CoreExpr] -> IO([Type])
 typecheckList exps = (mapM typecheck exps)
```

A.7 Eval.hs

```
-- An interpreter for F -- assumes that code has already been
-- typechecked
-- By Kirsten Chevalier
module Eval
where

import Table
import Core
import PrettyCore
```

```
import Monad
import List
import ThisUtil
import Result
import ThisUnique
import Build
import IOExts(fixIO)

-- May 26:
-- The interpreter goes into an infinite loop on
-- Test/Working/goal-inlined.hs if I uncomment the "say"s!
-- I wonder why?

-- For debugging
say expr = return()
--
-- putStrLn $ "evaluating " ++ (toString expr)  --
sayR res expr = return()
-- putStrLn $ "returning " ++ (showResult res) ++ ", result of evaluating "
--             ++ (pretty expr)
--

eval :: CoreExpr -> IO (Result)
eval expr = (eval' EmptyTable expr)

eval' :: Env -> CoreExpr -> IO (Result)
eval' env v@(Var id) = do
 say v
 (case (tableLookup id env) of
  (Just (ExprRes(expr, itsenv))) -> do
   res <- (eval' itsenv expr)
   sayR res v
   return res
  (Just val) -> do
   sayR val v
   return val
------------------ change this back
  Nothing -> (case (tableLookupString id funenv) of
      (Just res) -> do
                sayR res v
                return res
-- look these up by name, rather than unique, since they may have different
-- uniques in different programs
      Nothing -> (case (tableLookupString id specialenv) of
  (Just res) -> return res
  Nothing -> error $ "eval: unbound variable: "
                ++ (pretty (idName id)))))
eval' env ch@(Lit (MachChar c)) = do
 say ch
 let res = (CharRes c)
 sayR res ch
 return res
eval' env l@(Lit (MachInt i)) = do
```

```
 say l
 let res = (IntRes i)
 sayR res l
 return res
eval' env c@(Con con args) = do
   say c
-- Throw away type arguments
   newargs <- (mapM (eval' env) (filter (\ x -> (not (isTypeArg x))) args))
   let len = length newargs
   let res = (if (con == cons) then
      (if (len == 2) then
       listresCons (head newargs) (tail newargs)
       else
       error $ "eval: cons applied to " ++ (show len) ++
       " args, expects 2 args\n" ++ "offending expr = " ++
       (pretty c) ++ " whose args are " ++
       (showResult (ListRes newargs)))
      else if (con == tuple) then
      (if (len == 2) then
       (TupleRes ((head newargs),(head (tail newargs))))
       else
       error $ "eval: tuple applied to " ++ (show len)
       ++ " args, expects 2 args")
      else if (con == nil) then
      (if (len == 0) then
       (ListRes [])
       else
       error $ "eval: nil applied to " ++ (show len) ++
       " args, expects 0 args" ++ "offending expr = " ++
       (pretty c) ++ " whose args are " ++
       (showResult (ListRes newargs)))
      else if ((con == true') || (con == true))  then
      (if (len == 0) then
       (BoolRes True)
       else
       error $ "eval: true applied to " ++ (show len) ++
       " args, expects 0 args")
      else if ((con == false') || (con == false)) then
      (if (len == 0) then
       (BoolRes False)
       else
       error $ "eval: false applied to " ++ (show len) ++
       " args, expects 0 args")
      else
      error $ "eval: unknown constructor " ++ (pretty con))
   sayR res c
   return res
eval' env a@(App fun (Type _)) = do
  say a
  fun' <- eval' env fun
  sayR fun' a
-- Ignore type arguments
  return fun'
```

```
------------------- Special case for build
eval' env a@(App (App (Var funId) (Type _)) buildExpr) = do
  say a
  (if (funId == buildId) then
   do
   res <- (eval' env (App (App buildExpr
   (Lam x (Lam xs
   (Con cons [(Var x),(Var xs)]))))
           (Con nil [])))
   sayR res a
   return res
   else do
   res <- (evalFun env a)
   sayR res a
   return res)
eval' env a@(App fun arg) = do
  say a
  res <- evalFun env a
  sayR res a
  return res
eval' env l@(Lam arg body) = do
 say l
-- check if it's a type lambda; if so, just evaluate the body
 res <- (case (varDetails arg) of
 TyVar -> (eval' env body)
         _ -> return(Closure(arg, body, env)))
 sayR res l
 return res
eval' env l@(Let binds body) = do
 say l
-- Tried to do:
-- newenv <- (binds2env binds env)
-- but it doesn't work (causes an infinite loop). Not sure why.
 newenv <- (insertBinds (reverse (flattenBinds binds)) env)
 res <- (eval' newenv body)
 sayR res l
 return res
eval' env exp@(Case c var alts) = do
  say exp
  caseexp <- eval' env c
  let (con, vars, body) = findMatch caseexp alts
  res <-
    (eval'
     (tableInsert var caseexp
      (case caseexp of
       (ListRes [])     -> env
       (ListRes (x:xs)) -> (case con of
    (ConPat c) -> (if (c == cons) then
   (tableExtend vars
    [x, (ListRes xs)] env)
   else
   env)
    _ -> error $ "eval': this can't happen")
```

```
-- this case handles an arbitrary constructor of one argument;
-- more cases would have to be added here if other constructors
-- were added to the language
       _ -> (tableExtend vars [caseexp] env)))
     body)
  sayR res exp
  return res
eval' env exp = do
 say exp
 error $ "eval: attempt to evaluate " ++ (toString exp)


{-
binds2env :: CoreBind -> Env -> IO Env
-- extends <env> with the bindings in <bind>
binds2env (NonRec v expr) env = do
  newexpr <- eval' env expr
  return(tableInsert v newexpr env)
binds2env (Rec binds) env =
 let (vars, exprs) = unzip binds
 in do
    fixIO (\ newenv -> do
-- pretend we know the new environment and evaluate each binding in it
                       newexprs <- (mapM (\ expr -> do
                                                      eval' newenv expr) exprs)
-- then extend the old environment with the new bindings
                       return (tableExtend vars newexprs env)) -}

evalFun :: Env -> CoreExpr -> IO (Result)
evalFun env (App fun (Type _)) = (eval' env fun)
evalFun env a@(App fun arg) =  do
  res' <- (eval' env fun)
  case res' of
    (Closure (v, b, returnedEnv)) -> do
                                   newarg <- eval' env arg
                                   res <- (eval' (tableInsert v newarg
    returnedEnv) b)
     return res
    (CurriedPrim (primop, numargs, args)) -> do
       newarg <- eval' env arg
       (if (((length args) + 1) == numargs) then
   return(applyPrimop primop (args ++ [newarg]))
       else
   return(CurriedPrim(primop, numargs, args ++ [newarg])))
    exp -> error $ "eval: attempt to apply " ++ (showResult exp)


applyPrimop :: Primop -> [Result] -> Result
applyPrimop (primopId, fcn) args = (fcn args)

insertBinds :: [(Var, CoreExpr)] -> Env -> IO (Env)
insertBinds [] env = return env
insertBinds binds env =
```

```
-- this implements lazy evaluation (and lexical scope) for <let> expressions
 let (vars, exprs) = unzip binds
     newenv = (tableExtend vars (map (\ expr -> (ExprRes(expr, newenv))) exprs) env)
 in
    return newenv

flattenBinds :: Bind b -> [(b, Expr b)]
flattenBinds bnds = (foldr (\ bind rest ->
     (case bind of
      (NonRec var expr) -> (var, expr):rest
      (Rec binds) -> binds ++ rest))
      []
      [bnds])


findMatch :: Result -> [CoreAlt] -> CoreAlt
findMatch result alts = (case result of
 (ListRes l) -> (findMatchList l alts)
 (IntRes i) -> (findMatchLit result alts)
 (CharRes c) -> (findMatchLit result alts)
 (BoolRes b) -> (findMatchCon
 (if b then true' else false') alts)
 _ -> error $ "findMatch " ++ (showResult result))

findMatchList :: [Result] -> [CoreAlt] -> CoreAlt
findMatchList l alts = (case l of
[] -> (findMatchCon nil alts)
(x:xs) -> (findMatchCon cons alts))

findMatchCon :: Con -> [CoreAlt] -> CoreAlt
findMatchCon con alts =
    (case
     (filter
      (\ thing ->
       case thing of
       (DEFAULT, _, _)  -> True
       ((ConPat c), _, _) -> (conEquals c con)) alts) of
     [] -> error $ "findMatchCon: nonexhaustive patterns in case, con = "
            ++ (pretty con) ++ " pats're " ++
            (concat (map pretty (fst3 (unzip3 alts))))
     (alt:_) -> alt)

findMatchLit :: Result -> [CoreAlt] -> CoreAlt
findMatchLit litRes alts =
    (case (filter (\ thing -> case thing of
               ((LitPat lit), _, _) -> (litEquals lit litRes)
                                (DEFAULT, _, _) -> True
               ((ConPat con), _, _) ->
                  ((getUnique con) == (getUnique isharp))) alts) of
     [] -> error $ "findMatchLit: nonexhaustive patterns in case; pats're "
          ++ (concat (map pretty (fst3 (unzip3 alts)))) ++ " expr is " ++
          (showResult litRes)
     (alt:_) -> alt)
```

```
conEquals :: Con -> Con -> Bool
conEquals c1 c2 = ((c1 == true') && (c2 == true))
  || ((c1 == true) && (c2 == true'))
     || ((c1 == false) && (c2 == false'))
|| ((c1 == false') && (c2 == false))
   || (c1 == c2)

litEquals :: Literal -> Result -> Bool
litEquals (MachChar c1) (CharRes c2) = (c1 == c2)
litEquals (MachInt i1) (IntRes i2) = (i1 == i2)
litEquals _ _ = False

listresCons :: Result -> [Result] -> Result
listresCons thing1 [(ListRes [])] = (ListRes [thing1])
listresCons thing1 [(ListRes lst)] = (ListRes (thing1:lst))

------------------ definitions for Prelude ---------------------
true' = C Named{n_occ="True", n_uniq= 905969692}
false' = C Named{n_occ="False", n_uniq= 905969672}
listAlphaTy = TyConApp List [alphaTy]
listBetaTy = TyConApp List [betaTy]
f = (mkId (mkSysLocalName 1 "f") (FunTy alphaTy (FunTy betaTy betaTy)))
z = (mkId (mkSysLocalName 2 "z") betaTy)
l = (mkId (mkSysLocalName 3 "l") listAlphaTy)
foo = (mkId (mkSysLocalName 4 "foo") listAlphaTy)
x = (mkId (mkSysLocalName 5 "x") alphaTy)
xs = (mkId (mkSysLocalName 6 "xs") listAlphaTy)
myfoldr = (mkId (mkSysLocalName 805306375 "foldr")
   (ForAllTy alpha
    (ForAllTy beta
     (FunTy (FunTy alphaTy (FunTy betaTy betaTy))
      (FunTy betaTy (FunTy listAlphaTy betaTy))))))
mymap = (mkId (mkSysLocalName 805306488 "map")
 (ForAllTy alpha
  (ForAllTy beta
   (FunTy (FunTy alphaTy betaTy)
    (FunTy listAlphaTy listBetaTy)))))

idfun (result:_) = result
idfun l          = error $ "id applied to wrong number of args, namely"
                           ++ (concat (map showResult l))

------------ Prelude functions -- incomplete listing  -----------------------
minus = (mkId (mkSysLocalName 805306471 "-") (FunTy intTy (FunTy intTy intTy)))
equals = (mkId (mkSysLocalName 1627396023 "==") (FunTy intTy
 (FunTy intTy boolTy)))
times = (mkId (mkSysLocalName 1912602967 "*") (FunTy intTy
       (FunTy intTy intTy)))
equals2 = (mkId (mkSysLocalName 805306477 "==") (FunTy intTy
 (FunTy intTy boolTy)))
plus = (mkId (mkSysLocalName 1912602966 "+") (FunTy intTy (FunTy intTy intTy)))
mynot = (mkId (mkSysLocalName 1912602841 "not") (FunTy boolTy boolTy))
```

```
myor = (mkId (mkSysLocalName 1912602850 "||") (FunTy boolTy (FunTy boolTy boolTy)))
mynot2 = (mkId (mkSysLocalName 1912602843 "not") (FunTy boolTy boolTy))
times2 = (mkId (mkSysLocalName 1912602969 "*") (FunTy intTy (FunTy intTy intTy)))
plus2 = (mkId (mkSysLocalName 1912602968 "+") (FunTy intTy (FunTy intTy intTy)))

plus3 = (mkId (mkSysLocalName 1912602970 "+") (FunTy intTy (FunTy intTy intTy)))
append = (mkId (mkSysLocalName 805306370 "++")
  (ForAllTy alpha
   (FunTy listAlphaTy (FunTy listAlphaTy
       listAlphaTy))))
greaterthan = (mkId (mkSysLocalName 1912602857 ">") (FunTy intTy
     (FunTy intTy boolTy)))
or2 = (mkId (mkSysLocalName 1912602842 "||") (FunTy boolTy
      (FunTy boolTy boolTy)))
or3 = (mkId (mkSysLocalName 1912602840 "||") (FunTy boolTy
      (FunTy boolTy boolTy)))

cons1 = (mkId (mkSysLocalName (-1) "Cons")
 (ForAllTy alpha
  (FunTy alphaTy (FunTy listAlphaTy listAlphaTy))))
nil2 = (mkId (mkSysLocalName (-2) "Nil") (ForAllTy alpha listAlphaTy))

myand = (mkId (mkSysLocalName 1912602851 "&&") (FunTy boolTy (FunTy boolTy boolTy)))

funenv = (tableExtend
  [myfoldr, mymap, minus, equals, times, equals2,
           plus, mynot, myor, mynot2, times2,
           plus2,  plus3, append,  greaterthan, or2,
           or3, cons1, nil2, myand]
-- Some of these (foldr, map, cons, and nil) should have type arguments,
-- but since the interpreter ignores type arguments and applications anyway,
-- they are not specified.
  [(Closure
    (f,
    (Lam z
     (Lam l
      (Case (Var l) foo
       [(ConPat cons, [x, xs], (App (App (Var f) (Var x))
(App (App (App (Var myfoldr)
   (Var f)) (Var z))
 (Var xs)))),
(ConPat nil, [], (Var z))]))),
     EmptyTable)),
           (Closure (f,
    (Lam l (Case (Var l) foo
    [(ConPat nil, [], (Con nil [])),
     (ConPat cons,
      [x, xs],
      (Con cons [(App (Var f) (Var x)),
 (App (App (Var mymap) (Var f))
  (Var xs))]))])),
     EmptyTable)),
           (CurriedPrim
```

```
    ((minus, (\ lst ->
      (case lst of
       (res:(IntRes x):(IntRes y):[]) -> (IntRes (x - y))
       _ -> error $ "minus applied to wrong arg type, namely "
             ++ (showResult (ListRes lst))))), 3, [])),
           equalsDef,
   timesDef,
           equalsDef2,
   (CurriedPrim
    ((plus, (\ lst ->
     (case lst of
      (res:(IntRes x):(IntRes y):[]) -> (IntRes (x + y))
      _ -> error $ "plus applied to wrong arg type, namely "
            ++ (showResult (ListRes lst))))), 3, [])),
   (CurriedPrim
    ((mynot, (\ lst ->
      (case lst of
       ((BoolRes x):[]) -> (BoolRes (not x))
                       _ -> error "not applied to wrong arg type"))), 1, [])),
           (CurriedPrim
    ((myor, (\ lst ->
     (case lst of
      ((BoolRes x):(BoolRes y):[]) -> (BoolRes (x || y))
      _ -> error $ "or applied to wrong arg type, namely "
                   ++ (showResult (ListRes lst))))), 2, [])),
   (CurriedPrim
    ((mynot2, (\ lst ->
       (case lst of
((BoolRes x):[]) -> (BoolRes (not x))
                        _ -> error "not applied to wrong arg type"))), 1, [])),
   (CurriedPrim
    ((times2, (\ lst ->
       (case lst of
(res:(IntRes x):(IntRes y):[]) -> (IntRes (x * y))
    _ -> error $ "* applied to wrong arg type, namely "
        ++ (showResult (ListRes lst))))), 3, [])),
   (CurriedPrim
    ((plus2, (\ lst ->
      (case lst of
       (res:(IntRes x):(IntRes y):[]) -> (IntRes (x + y))
       _ -> error $ "plus applied to wrong arg type, namely "
             ++ (showResult (ListRes lst))))), 3, [])),
   (CurriedPrim
    ((plus3, (\ lst ->
      (case lst of
       (res:(IntRes x):(IntRes y):[]) -> (IntRes (x + y))
       _ -> error $ "plus applied to wrong arg type, namely "
                     ++ (showResult (ListRes lst))))), 3, [])),
   (CurriedPrim
    ((append, (\ lst ->
       (case lst of
((ListRes l1):(ListRes l2):[]) -> (ListRes (l1 ++ l2))
_ -> error "++ applied to wrong arg type"))), 2, [])),
```

```
   (CurriedPrim
    ((greaterthan, (\ lst ->
    (case lst of
     (res:(IntRes x):(IntRes y):[]) -> (BoolRes (x > y))
     _ -> error "> applied to wrong arg type"))), 3, [])),
   (CurriedPrim
    ((or2, (\ lst ->
    (case lst of
     ((BoolRes x):(BoolRes y):[]) -> (BoolRes (x || y))
     _ -> error $ "or applied to wrong arg type, namely "
                  ++ (showResult (ListRes lst))))), 2, [])),
  (CurriedPrim
   ((or3, (\ lst ->
   (case lst of
    ((BoolRes x):(BoolRes y):[]) -> (BoolRes (x || y))
    _ -> error $ "or applied to wrong arg type, namely " ++
                   (showResult (ListRes lst))))), 2, [])),
  (CurriedPrim
   ((cons1, (\ lst ->
     (case lst of
      (x:(tl@(ListRes l)):[]) -> (listresCons x [tl])
      _ -> error "Cons applied to wrong arg type"))), 2, [])),
  (ListRes []),
  (CurriedPrim
   ((myand, (\ lst ->
     (case lst of
      ((BoolRes x):(BoolRes y):[]) -> (BoolRes (x && y))
      _ -> error "&& applied to wrong arg type"))), 2, []))]
  EmptyTable)

equalsDef = (CurriedPrim
     ((equals, (\ lst ->
(case lst of
 ((IntRes x):(IntRes y):[]) -> (BoolRes (x == y))
 _ -> error $ "equals applied to wrong arg type, namely " ++ (showListRes lst)))),
      2, []))

equalsDef2 = (CurriedPrim
      ((equals2, (\ lst ->
  (case lst of
   (res:(IntRes x):(IntRes y):[]) -> (BoolRes (x == y))
   _ -> error "equals applied to wrong arg type (Def2)"))),
       3, []))

timesDef = (CurriedPrim
    ((times,
      (\ lst ->
       (case lst of
(res:(IntRes x):(IntRes y):[]) -> (IntRes (x * y))
_ -> error $ "*1912602967 applied to wrong arg type, namely "
           ++ (showResult (ListRes lst))))), 3, []))

------------- junk that GHC dumps in ------------------------
```

```
fnumint = (mkId (mkSysLocalName 1912603492 "$fNumInt") (FunTy intTy intTy))
fromint = (mkId (mkSysLocalName 805306469 "fromInt") (FunTy intTy intTy))
isharp = (mkId (mkSysLocalName 905969676 "I#") (FunTy intTy intTy))
fnuminteger = (mkId (mkSysLocalName 1912603535 "$fNumInteger") intTy)
dnum = (mkId (mkSysLocalName 1627396004 "$dNum") (FunTy intTy intTy))
csharp = (mkId (mkSysLocalName 905969667 "$wC#") (FunTy charTy charTy))
feqint = (mkId (mkSysLocalName 1912603385 "$fEqInt") (FunTy intTy intTy))
fordint = (mkId (mkSysLocalName 1912603311 "$fOrdInt") (FunTy intTy intTy))
fintegralint = (mkId (mkSysLocalName 1912604395 "$fIntegralInt")
(FunTy intTy intTy))

specialenv = (tableExtend [feqint, fnumint, fromint, fnuminteger,
  dnum, csharp, fordint, fintegralint, isharp]
     (take 9 (repeat idClosure))
     EmptyTable)
idClosure = (Closure (x, (Var x), EmptyTable))
-------------------------------------------------------------
```

A.8 Table.hs

```
-- implements a table as a list of pairs
-- By Kirsten Chevalier
module Table
    where

    import List
    import Core(Var, varName, n_occ)
    import PrettyCore

    data Show b => Table b = T [(Var, b)] | EmptyTable

    table :: Table b -> [(Var, b)]
    table (T t) = t
    table EmptyTable = []
    toTable :: Show b => [(Var, b)] -> Table b
    toTable [] = EmptyTable
    toTable t = T t

    tableLookup ::  Var -> Table b -> Maybe b
    tableLookup v t = let findRes = (find (\ (vr, ty) -> (vr == v)) (table t))
  in
  (case findRes of
   Nothing -> Nothing
   Just(_, res) -> Just(res))

    tableLookupString ::  Var -> Table b -> Maybe b
    tableLookupString v t = let findRes = (find (\ (vr, ty) ->
 ((n_occ(varName vr)) == (n_occ(varName v))))
   (table t))
in
(case findRes of
 Nothing -> Nothing
```

```
 Just(_, res) -> Just(res))

-- if a value is inserted that is already in the table, remove the previous value

    tableInsert :: (Show b) => Var -> b -> Table b -> Table b
    tableInsert v ty t = (toTable ((v, ty):(filter (\ (var, typ) -> (var /= v))
    (table t))))

    tableExtend ::  (Show b) => [Var] -> [b] -> Table b -> Table b
    tableExtend vs tys t = (foldr (\ (v, ty) rest -> (tableInsert v ty rest))
    t
    (zip vs tys))

    tableDelete :: (Show b) => Var -> Table b -> Table b
    tableDelete v t = (toTable (filter (\ (vr, ty) -> (vr /= v)) (table t)))

    showTable :: (Show b) => Table b -> String
    showTable binds = ("---BEGIN ENV------\n" ++
       (foldr
(\ (var, expr) rest ->
 (pretty var) ++ " = " ++ (show expr) ++  "\n" ++ rest)
""
(table binds)) ++
       "---END ENV--------\n")

    tableMerge :: (Show b) =>  [Table b] -> Table b
    tableMerge tables = (toTable (remDups (concat (map table tables))))

    remDups :: [(Var, b)] -> [(Var, b)]
    remDups pairs = (foldr (\ p@(var, ty) rest ->
    (case (find (\ (v,_) -> (v == var)) rest) of
     Just _  -> rest
     Nothing -> p:rest))
     []
     pairs)
```

A.9 Result.hs

```
-- provides Result type, representing the result of an evaluation
-- By Kirsten Chevalier
module Result
where

import Core
import PrettyCore
import Table

type Env = Table Result
```

```
data Result =
   IntRes Integer
 | CharRes Char
 | ListRes [Result]
 | BoolRes Bool
 | TupleRes (Result, Result)
-- (\ v -> expr), defined in <env>, is represented as
-- Closure(v, expr, <env>)
 | Closure (Var, CoreExpr, Env)
 | CurriedPrim (Primop, Int, [Result])
 | ExprRes (CoreExpr, Env)

instance Show Result where
  show r = showResult r

instance Eq Result where
  (IntRes i) == (IntRes j) = i == j
  (CharRes c1) == (CharRes c2) = c1 == c2
  (ListRes l1) == (ListRes l2) = l1 == l2
  (BoolRes b1) == (BoolRes b2) = b1 == b2
  (TupleRes t1) == (TupleRes t2) = t1 == t2
-- should this really check whether the environments are the same, too?
-- Probably. It's also probably wrong anyway.
  (Closure (v1, body1, _)) == (Closure (v2, body2, _)) = (v1 == v2)
      && (cheapEqExpr body1 body2)
  (CurriedPrim ((nm1, _), _, _)) == (CurriedPrim ((nm2, _), _, _)) = nm2 == nm1
  (ExprRes (e1, _)) == (ExprRes (e2, _)) = (cheapEqExpr e1 e2)
  _ == _ = False

type Primop = (Var, [Result] -> Result)

showResult :: Result -> String
showResult (IntRes i) = (show i)
showResult (CharRes c) = (show c)
showResult (ListRes []) = "[]"
showResult (ListRes l) = ("[" ++ (showListRes l) ++ "]")
showResult (BoolRes b) = (show b)
showResult (TupleRes (res1,res2)) =
    ("(" ++ (showResult res1) ++ " , " ++ (showResult res2) ++ ")")
showResult (Closure (v, exp, env)) = (showClosure (Lam v exp) env)
showResult (CurriedPrim ((primopId, _), _, args)) =
    ("primop: (" ++ (pretty primopId) ++ " " ++
     (showResult (ListRes args)) ++ ")")
showResult (ExprRes (expr, env)) = (showClosure expr env)

showClosure exp env = ((toString exp) ++ "\n       env = " ++ (showTable env))

showListRes l = ((concat ((map (\ x -> ((showResult x) ++ ", "))
   (take ((length l) - 1) l))) ++ showResult (last l)))

toString expr = (pretty expr)
```

A.10 F2Haskell.hs

```
-- F to Haskell translator
-- By Kirsten Chevalier
module F2Haskell
where

  import Core
  import PrettyCore
  import Eval
  import ThisUnique
  import ThisUtil
  import Table
  import Result

  first :: [a] -> a
  first (x:xs) = x

  second :: [a] -> a
  second (x:(y:xs)) = y

  parens :: String -> String
  parens s = "(" ++ s ++ ")"

  curlies :: String -> String
  curlies s = "{" ++ s ++ "}"

  sep :: String -> String -> String -> String
  sep separator firstString secondString =
      firstString ++ separator ++ secondString

  var2String :: Var -> String
  var2String var = (case (preludeLookup var) of
 -- for Prelude functions, don't print the unique
    Just str -> str
    Nothing -> (pretty var))

  -- given an F expression, returns a string representing the corresponding
  -- Haskell program
  f2haskell :: CoreExpr -> String

  f2haskell (Let binds body) = "main = " ++ (f2haskell' body) ++ "\n\n" ++
   (concat (convertBinds binds))
  f2haskell _ = error $ "f2haskell: Let expected"

  f2haskell' (Var v) = var2String v
  f2haskell' (Lit (MachChar c)) = "'" ++ [c] ++ "'"
  f2haskell' (Lit (MachInt i)) = (show i)
  f2haskell' (Con con args) = (if (con == cons) then
      (parens (sep ":"
       (f2haskell' (first args'))
       (f2haskell' (second args'))))
      else if (con == nil) then
```

```
      "[]"
      else if (con == tuple) then
      (parens (sep ","
       (f2haskell' (first args'))
       (f2haskell' (second args'))))
      else if ((con == true) || (con == true')) then
      "True"
      else if ((con == false) || (con == false')) then
      "False"
      else if ((getUnique con) ==
       (varUnique isharp)) then
      (f2haskell' (first args'))
      else
      error $ "bad constructor" ++ (pretty con))
                            where (_, args') = span isTypeArg args
  f2haskell' (App fun (Type _)) = (f2haskell' fun)
  f2haskell' (App fun arg) = (parens (sep " "
     (f2haskell' fun)
     (f2haskell' arg)))
  f2haskell' (Lam var body) = (case (idType var) of
       (TyConApp Kind _) -> (f2haskell' body)
       _ -> (parens ("\\ " ++ (sep " -> "
       (var2String var)
       (f2haskell' body)))))
  f2haskell' (Let binds expr) =
      (parens (case flat of
       [] -> (f2haskell' expr)
       _ -> (sep "in "
     ("let " ++
      (curlies
       (concat
((map (bind2haskell True)
  (allbutlast flat))
 ++ [(bind2haskell False (last flat))]))))
       (f2haskell' expr))))
 where flat = flattenBinds binds
  f2haskell' (Case expr _ alts) =
      (parens ("case " ++ (f2haskell' expr) ++ " of\n"
       ++ (concat ((map
    (alt2haskell True)
    (allbutlast alts)) ++
   [(alt2haskell False (last alts))]))))
  f2haskell' (Type ty) = (type2haskell ty)  -- this shouldn't be necessary

  bind2haskell :: Bool -> (Var, CoreExpr) -> String
  bind2haskell semi (var, expr) = ((sep " = "
    (var2String var)
    (f2haskell' expr)) ++
   (if semi then
    ";\n"
    else
    "\n\n"))
```

```
alt2haskell :: Bool -> (Pat, [Var], CoreExpr) -> String
alt2haskell newline (pat, vars, expr) =
  ((sep " -> "
    (pat2string pat vars)
    (f2haskell' expr)) ++ (if newline then
   "\n"
   else
   ""))

pat2string :: Pat -> [Var] -> String
pat2string (ConPat con) vars = (f2haskell' (Con con (map Var vars)))
pat2string (LitPat lit) _ = (f2haskell' (Lit lit))
pat2string DEFAULT _ = "_"

type2haskell :: Type -> String
type2haskell (ForAllTy tyvar ty) = type2haskell ty
type2haskell (FunTy ty1 ty2) = (sep " -> "
(type2haskell ty1)
(type2haskell ty2))
type2haskell (TyConApp tc tys) = (parens (sep " "
  (show tc)
  (concat (map (\ s -> s ++ " ")
   (map type2haskell tys)))))
type2haskell (TyVarTy v) = (var2String v)

convertBinds binds = (map (bind2haskell False) (removeGarbage (flattenBinds binds)))

allbutlast l = (take ((length l) - 1) l)

removeGarbage binds = (filter (\ (var, expr) ->
((not (containsDollarSign (Var var))) &&
 (not (isDuplicate (Var var)))))
binds)

containsDollarSign (Var v) = (first (n_occ (varName v))) == '$'
containsDollarSign expr = mapExprBool containsDollarSign expr

isDuplicate (Var v) = (case (preludeLookup v) of
Just _ -> True
Nothing -> False)

preludeLookup :: Var -> Maybe(String)
preludeLookup var = (case (tableLookupString var preludeFuns) of
     Just (ListRes s) -> (Just (resToString s))
     Nothing -> (case (tableLookupString var specialenv) of
  Just _ -> Just "id"
  Nothing -> Nothing))

preludeFuns = (tableExtend
   [myfoldr, mymap, minus, equals, times,
    plus, myor, append, greaterthan,
    mynot, myand, cons1, nil2]
   (map (\ l -> (ListRes (map CharRes l)))
```

```
      ["foldr", "map", "(-)", "(==)", "(*)",
       "(+)", "(||)", "(++)", "(>)", "not",
       "(&&)", "(:)", "[]"])
     EmptyTable)

  resToString chars = (map (\ (CharRes c) -> c) chars)
```

A.11 TypeInference.hs

```
-- The main list abstraction algorithm
-- By Olaf Chitil, with minor changes by Kirsten Chevalier
module TypeInference where

import IOExts ( IORef, newIORef, readIORef, writeIORef )
import Monad(MonadPlus(..),guard,msum,filterM,liftM)
import MonadTransformer
import ThisUnique
import Core
import ThisUtil
import List ((\\))
import Maybe(isJust, fromJust, fromMaybe)
import IOExts( unsafePerformIO )
import PrettyCore
import CountIO
import Build

{-
The Type inference monad
- for keeping track of continously modified types
- may fail
-}

noFailTIM :: TIM a -> IO a
noFailTIM (MT m) = do justx <- m; return (fromJust justx)

{-
functions for creating new type variables and reading and writing them
in the monad
-}

-- newTITyVar :: IO Type
-- newTITyVar = do
--   tyVar <- newMutTyVar (mkSysLocalName 0 "t") (TyConApp Kind [])
--   return $ TyVarTy tyVar

newTITyVar :: CountIO Type
newTITyVar = do
  count <- getCounter
  tyVar <- lift $ newMutTyVar (mkSysLocalName count "t") (TyConApp Kind [])
  incCounter
  return $ TyVarTy tyVar
```

```
readTITyVar :: TyVar -> IO (Maybe Type)
readTITyVar = readMutTyVar

writeTITyVar :: TyVar -> Type -> IO ()
writeTITyVar tyvar ty = writeMutTyVar tyvar (Just ty)

sameTITyVar :: TyVar -> TyVar -> Bool
sameTITyVar VarG{varDetails=MutTyVar ref1 _} VarG{varDetails=MutTyVar ref2 _} =
  ref1 == ref2

{-
The actual type inference algorithm for the simply typed language
-}

tyInfer :: CoreExpr -> Type -> TIMC ()

tyInfer (Lit lit) ty = do
  tim2timc (unify (litType lit) ty)
tyInfer (Var id) ty = do
  tim2timc (unify (idType id) ty)
tyInfer (Con con args) ty = do
  let (tyArgs,exprArgs) = span isTypeArg args
      tys = map (\(Type ty) -> ty) tyArgs
  exprArgTypes <- sequence (map tyInferArg exprArgs)
  tim2timc ((unify (applyTys (conType con) tys)
    (foldr mkFunTy ty exprArgTypes)))
  where
  tyInferArg :: CoreExpr -> TIMC (Type)
  tyInferArg expr = do
    tyVar <- lift newTITyVar
    tyInfer expr tyVar
    return tyVar
tyInfer a@(App fun (Type argTy)) ty = do
    tyVar <- lift newTITyVar
    tyInfer fun tyVar
    forAllTy <- io2timc $ elimTypeBounds tyVar
    tim2timc (unify ty (applyTy forAllTy argTy))
tyInfer a@(App (App (Var id) (Type argTy)) arg) ty =
-- Special case for build
  (if (id == buildId) then do
    tim2timc (unify (TyConApp List [argTy]) ty)
    tyInfer arg (ForAllTy beta (FunTy (FunTy argTy (FunTy betaTy betaTy))
      (FunTy betaTy betaTy)))
   else
    tyInferApp a ty)
tyInfer a@(App fun arg) ty = tyInferApp a ty
tyInfer (Lam var expr) ty = do
  tyVar <- lift newTITyVar
  if isTyVar var then
   tim2timc (unify (ForAllTy var tyVar) ty)
   else -- id is term variable
   tim2timc (unify (mkFunTy (idType var) tyVar) ty)
```

```
  tyInfer expr tyVar
tyInfer (Let (NonRec id expr) body) ty = do
  tyInfer body ty
  tyInfer expr (idType id)
tyInfer (Let (Rec binds) body) ty = do
  tyInfer body ty
  mapM_ (\(id,expr) -> tyInfer expr (idType id)) binds
tyInfer (Case expr id alts) ty = do
  tyInfer expr scrutineeTy
  tyVars <- lift $ sequence $ replicate numOfTyVars newTITyVar
  --- mapM_           :: Monad m => (a -> m b) -> [a] -> m ()    -- from Prelude
  mapM_ (tyInferAlt tyVars) alts
  where
  scrutineeTy = idType id
  numOfTyVars
    | null alts = 0
    | otherwise =
      length $ fst $ splitForAllTys $ patType $ (\(c,_,_) -> c) $ head alts
  tyInferAlt tyVars (pat, ids, expr) = do
    tim2timc (unify (applyTys (patType pat) tyVars)
      (foldr mkFunTy scrutineeTy $ map idType ids))
    tyInfer expr ty
    return ()

tyInferApp (App fun arg) ty =
    do -- argument is term expression, not type
tyVar <- lift newTITyVar
tyInfer fun (mkFunTy tyVar ty)
tyInfer arg tyVar

{-
Unfication of types
WITHOUT OCCUR CHECK! (have to prove that it is not necessary)
Note that only flexiTyVars can be bound,
the others (TyVar) are handled like constants.

Strange results for some inputs (d type inference variable):
unify (forall b. d -> d) (forall a. a -> d) = forall b. a -> a
Such inputs do not occure here.
-}

unify :: Type -> Type -> TIM ()

unify type1 type2 =
  let
    ty1 = unsafePerformIO (elimTypeTyVars type1)
    ty2 = unsafePerformIO (elimTypeTyVars type2)
  in
   unify emptyUFM type1 type2
  where
  -- Use finite map to implement comparison modulo alpha conversion
  -- for forall bound type variables
  unify :: UniqFM (TyVar) -> Type -> Type -> TIM ()
```

```
  unify alpha f1@(FunTy arg1 res1) f2@(FunTy arg2 res2) = do
    unify alpha arg1 arg2
    unify alpha res1 res2
  unify alpha (TyConApp tycon1 args1) (TyConApp tycon2 args2) = do
    guard (tycon1 == tycon2) -- then also length args1 == length args2
    sequence_ (zipWith (unify alpha) args1 args2)
  unify alpha t@(TyVarTy tyVar1) type2 | isMutTyVar tyVar1 = do
    contTyVar1 <- lift $ readTITyVar tyVar1
    case contTyVar1 of
      Nothing    -> lift $ bindTyVar tyVar1 type2
      Just type1 -> unify alpha type1 type2
  unify alpha type1 t@(TyVarTy tyVar2) | isMutTyVar tyVar2 = do
    content <- lift $ readTITyVar tyVar2
    case content of
      Nothing    -> lift $ writeTITyVar tyVar2 type1
      Just type2 -> unify alpha type1 type2
  unify alpha t1@(TyVarTy tyVar1) t2@(TyVarTy tyVar2) =  -- both non-mutable tyvars
    guard (fromMaybe tyVar1 (lookupUFM alpha tyVar1) == tyVar2)
  unify alpha (ForAllTy tyVar1 ty1) (ForAllTy tyVar2 ty2) =
    unify (addToUFM alpha tyVar1 tyVar2) ty1 ty2
  unify alpha type1 type2 = error ("Unification failed: " ++ show (ppr False type1)
                      ++ " : " ++ show (ppr False type2))

{-
Unification of an unbound type variable with a type always succeeds,
but have to take care not to create a cyclic structure.
Also avoid chains of type variables as far as easily achievable.
-}

bindTyVar :: TyVar -> Type -> IO ()

bindTyVar tyVar1 type2@(TyVarTy tyVar2) | isMutTyVar tyVar2 = do
  contTyVar2 <- readTITyVar tyVar2
  case contTyVar2 of
    Nothing ->
      if sameTITyVar tyVar1 tyVar2 then
        return () -- don't create a cycle
       else
        writeTITyVar tyVar1 type2
    Just contTy2 -> bindTyVar tyVar1 contTy2  -- no unneccessary chains
bindTyVar tyVar1 type2 = writeTITyVar tyVar1 type2

{-
Monads for analysing a producer and inlining binding.
Uses type inference and may fail any time, hence requires TIM
Furthermore threads set of all used inlineable ids.
-}

type InlineEnv = UniqFM CoreExpr -- body of an inlineable id
type InInst = UniqFM Id  -- instance id of an id
type InstEnv = UniqFM (CoreExpr, CoreExpr, Id)
               -- (original expr, renamed body, instance id)
```

```
type AnM a = StateT (InInst, InstEnv, UniqSupply) (MaybeT IO) a
                                                     -- TIM
type InM a = StateT (InInst, InstEnv, UniqSupply) IO a


type ConsToPlaceholders = Unique -> Maybe (Type -> CoreExpr)
type PlaceholdersToCons = Id -> IO (Maybe (CoreExpr))

{-
Perform analysis of an expression.
Needs bindings which may be inlined.
-}

analyse :: ConsToPlaceholders -> Type -> InlineEnv
           -> CoreExpr -> Type -> AnM CoreExpr

analyse consToPlaceholders tyToRemove inlineEnv
  expr desiredType = do
    (newInstEnv, instExpr) <- liftInside $
      instantiate inlineEnv tyToRemove expr
    (_,oldInstEnv,_) <- readState
    ((instEnv, tyVarExpr), cnt) <- lift $ lift $
       (runCounterWith
(replaceExprTypes tyToRemove oldInstEnv newInstEnv instExpr) 0)
    updateState (\(inInst,_,us)->(inInst,instEnv,us))
    (placeholderTyVarExpr, cnt1) <- lift $ lift $
      (runCounterWith
       (replaceConstructors consToPlaceholders tyToRemoveArgs tyVarExpr) cnt)
    (res, cnt2) <- lift $ lift $ runCounterWith
    (runMaybeT $ (tyInfer placeholderTyVarExpr desiredType)) cnt1
    return placeholderTyVarExpr
  where
    tyToRemoveArgs =
case tyToRemove of
 TyConApp tcon targs -> targs
 otherwise           ->
     error $ "TypeInference.analyse: tyToRemove is " ++
   pretty tyToRemove ++
   ", tyConApp expected"

{-
Apply monadic function repeatedly until test returns True
-}

untilM :: Monad m => (a -> Bool) -> (a -> m a) -> m a -> m a
untilM test f start = do
  x <- start
  if test x then
    return x
   else
    untilM test f (f x)

{-
```

```
Determine and analyse all bindings which need to be inlined
according to the given test function.
Needs all bindings which may be inlined.
-}

inline :: ConsToPlaceholders -> Type -> InlineEnv -> (CoreBndr -> IO Bool)
          -> AnM [(CoreBndr, CoreExpr)]

inline consToPlaceholder typeToRemove inlineEnv hasToBeInlined = do
  (inlinedIds, inlinedExprs, _) <-
    untilM (null . thrd3) inlineOnce (liftInside start)
  return (zip inlinedIds inlinedExprs)
  where
  start :: InM ([CoreBndr],  [CoreExpr], [CoreBndr])
  start = do
    idsToBeInlined <- detIdsToBeInlined hasToBeInlined []
    return ([], [], idsToBeInlined)
  -- extend inlined bindings by those given in the 3rd component
  -- and determine list of ids which need to be inlined next
  inlineOnce :: ([CoreBndr], [CoreExpr], [CoreBndr])
                -> AnM ([CoreBndr],  [CoreExpr], [CoreBndr])
  inlineOnce (inlinedIds, inlinedExprs, idsNextToBeInlined) = do
    newInlinedExprs <-
      analyseIds consToPlaceholder typeToRemove inlineEnv idsNextToBeInlined
    newIdsNextToBeInlined <-
      liftInside $ detIdsToBeInlined hasToBeInlined allInlinedIds
    return ( allInlinedIds
           , newInlinedExprs ++ inlinedExprs
           , newIdsNextToBeInlined )
    where
    allInlinedIds = idsNextToBeInlined ++ inlinedIds

-- analyse expressions which are bound to the given ids
analyseIds :: ConsToPlaceholders -> Type -> InlineEnv
              -> [CoreBndr] -> AnM [CoreExpr]
analyseIds consToPlaceholder typeToRemove inlineEnv idsToBeInlined = do
  (_, instEnv, _) <- readState
  let exprsToBeInlined = map (snd3 . fromJust . lookupUFM instEnv)
                             idsToBeInlined
  sequence $ zipWith (analyse consToPlaceholder typeToRemove inlineEnv)
    exprsToBeInlined (map idType idsToBeInlined)


-- determine which of the given ids need to be inlined
detIdsToBeInlined :: (CoreBndr -> IO Bool)
                     -> [CoreBndr] -> InM [CoreBndr]

detIdsToBeInlined hasToBeInlined alreadyInlined = do
  (_, instEnv, _) <- readState
  tobeInlinedIds <- lift $
    filterM hasToBeInlined $ map thrd3 $ eltsUFM instEnv
  return (tobeInlinedIds \\ alreadyInlined)
```

```
{-
Test if a unique occurs as type variable unique
-}

occursInTIBndr :: Unique -> CoreBndr -> IO Bool

occursInTIBndr unique id = do
  let ty = varType id
  ty' <- elimTypeTyVars ty
  return (occursInType unique ty')


occursInType :: Unique -> Type -> Bool

occursInType unique (TyVarTy tyVar) = unique == realUnique tyVar
occursInType unique (TyConApp _ args) = any (occursInType unique) args
occursInType unique (FunTy arg res) =
  occursInType unique res || occursInType unique arg -- try result first
occursInType unique (ForAllTy var body) = occursInType unique body



{-
Try to convert an expression of type list into build form.
Needs bindings whose' types contain lists and which may be inlined.
-}

typeOf :: CoreExpr -> Type
typeOf (Lit (MachInt x))      = (TyConApp Int [])
typeOf (Lit (MachChar x))     = (TyConApp Char [])
typeOf (Var id)               = (last (getTypes (idType id)))
typeOf (Con con args)         =  mkFunTys (drop 1 resTy) (last resTy)
                                  where
                                  (tyArgs, exprArgs) = span isTypeArg args
          tys = map (\ (Type ty) -> ty) tyArgs
                                  conTy = (applyTys (conType con) tys)
                                  resTy = (drop ((length args) - 1) (getTypes conTy))
typeOf (App fun (Type argTy)) = (applyTy (typeOf fun) argTy)
typeOf (App fun arg)          = mkFunTys (drop 2 funTy) (last funTy)
                                 where
                                  funTy = (getTypes (typeOf fun))
typeOf (Lam var expr)         = (typeOf expr)
typeOf (Let _ body)           = (typeOf body)
typeOf (Case expr id (a:as))  = case a of
                                 (con, args, exp) -> (typeOf exp)
typeOf foo =  error ("typeOf: " ++ (pretty foo))

-- given t_0 -> t_1 -> ... -> t_n, returns [t_0, t_1, ... , t_n]
-- does the same for (forall t_0 (forall t_1 ... ))
getTypes :: Type -> [Type]
getTypes (FunTy arg res) = arg:(getTypes res)
getTypes x = [x]

-- Made this a global definition and not a local one, so we can use
```

```
-- it in Examples
buildTyVar = VarG{ varName = Named "build" 101
                   , varType = TyConApp Kind []
                   , realUnique = 101, varDetails = TyVar {-Build typeToRemove-}}
nilPlaceholderUnique = 3
consPlaceholderUnique = 4

buildListBuild :: UniqSupply -> InlineEnv -> CoreExpr -> IO (Maybe CoreExpr)

buildListBuild us ie ce = do
  (buildBuild typeToRemove buildTyVar consToPlaceholders placeholdersToCons us ie ce)
 where
  typeToRemove = (typeOf ce)
  typeToRemoveUnlisted = (case typeToRemove of
  (TyConApp List [x]) -> x
  ty -> ty)

  consToPlaceholders = lookupUFM_Directly consToPlaceholdersUFM
  consToPlaceholdersUFM = listToUFM
    [ (nilDataConKey, \ty -> Var (mkSysLocal "n" nilPlaceholderUnique ty))
    , (consDataConKey, \ty -> Var (mkSysLocal "c" consPlaceholderUnique
        (FunTy typeToRemoveUnlisted  (FunTy ty ty))))
    ]
  placeholdersToCons id =
    case lookupUFM placeholdersToConsUFM id of
      Just cons -> do
        test <- hasBuildTyVar id
        return $ if test then Nothing else Just cons
      Nothing   -> return Nothing

  placeholdersToConsUFM = listToUFM
    [ (nilPlaceholderUnique, mkApps (Var nilId) [Type typeToRemoveUnlisted])
    , (consPlaceholderUnique, mkApps (Var consId) [Type typeToRemoveUnlisted])
    ]
  nilId = mkVanillaId nilName (dataConType $ C nilName)
  consId = mkVanillaId consName (dataConType $ C consName)
  nilName = Named{n_occ="Nil", n_uniq=nilDataConKey}
  consName = Named{n_occ="Cons", n_uniq=consDataConKey}
  hasBuildTyVar :: CoreBndr -> IO Bool
  hasBuildTyVar = occursInTIBndr (getUnique buildTyVar)
  boolTy = TyConApp Bool []

{-
Build inference with inlining.
Analyse an expression and inline bindings when necessary.
Needs bindings which may be inlined.
-}

buildBuild :: Type -> TyVar
              -> ConsToPlaceholders
              -> PlaceholdersToCons
              -> UniqSupply
              -> InlineEnv -> CoreExpr -> IO (Maybe CoreExpr)
```

```
buildBuild typeToRemove buildTyVar consToPlaceholder placeholdersToCons
  uniqSupply inlineEnv expr = runMaybeT $ do
    ((_, finalInstEnv,_) , analysedExpr) <- runStateMT (do
      analysedExpr <-
        analyse consToPlaceholder typeToRemove inlineEnv expr newType
      analysedBinds <-
        inline consToPlaceholder typeToRemove inlineEnv hasToBeInlined
      return $ Let (Rec analysedBinds) analysedExpr
      )
      (emptyUFM, emptyUFM, uniqSupply)
    analExprForOutput <- lift $ elimExprTyVars analysedExpr
    polyTyExpr <- lift $ reInsertConstructors placeholdersToCons analysedExpr
    polyTyExpr2 <- lift $ elimExprBoundTyVars typeToRemove polyTyExpr
    -- replace all type inference variables which still occur by the
    -- type of the intermediate data structure
    let polyTyExpr3 = uninstantiate finalInstEnv buildTyVar polyTyExpr2
    return polyTyExpr3
    where
    hasToBeInlined = occursInTIBndr buildTyVarUnique
    buildTyVarUnique = getUnique buildTyVar
    newType = TyVarTy buildTyVar


{-
Replaces every list type by a new type variable in the types of all
term identifiers which are locally bound or are externally bound
and can be inlined so that they can be converted as well.
(e.g. no data constructors or canonical identifiers of data constructors)

Hence the transformation needs a set of possibly convertible
external term identifiers.

Besides the term with replaced list types the transformation returns
a set of modified external identifiers. These are those external identifiers
which occur in the input term and who's list types were replaced.

List type replacement takes only place at a binding occurrence or
in the set of external identifiers.
At all other occurrences an identifier is replaced by its respective
binding occurrence (comparison by unique), so all occurrences are shared.

Assume that the type to be replaced has an algebraic data type constructor
at the top (no function type or type variable).

The algorithm threads a set of external term identifiers who's types
have been modified.
-}

replaceExprTypes ::
  Type -> InstEnv -> InstEnv -> CoreExpr -> CountIO (InstEnv, CoreExpr)

replaceExprTypes typeToReplace oldInstEnv newInstEnv expr = do
```

```
  let (orgUniqs, instInfs) = unzip $ ufmToList newInstEnv
      (olds, bodies, ids) = unzip3 instInfs
  repIds <- mapM (replaceIdTypes typeToReplace) ids
  let instEnv = addListToUFM oldInstEnv $ zip orgUniqs $
                zip3 olds bodies repIds
  repExpr <- rep instEnv expr
  return (instEnv, repExpr)
  where
  rep :: InstEnv -> CoreExpr -> CountIO CoreExpr
  rep instEnv expr = replace emptyUniqSet expr
    where
    replace :: UniqSet Id -> CoreExpr -> CountIO CoreExpr
    -- the environment stores the locally bound term variables
    -- it is only passed downwards
    replace env expr@(Var id) =
      case lookupUniqSet env id `mplus`
           fmap (\(_,_,x)->x) (lookupUFM instEnv id) of
        Just repId -> return (Var repId)
        Nothing    -> return expr
    replace env (Lam id body) = do
      repId <- replaceIdTypes typeToReplace id
      repBody <- replace (addOneToUniqSet env repId) body
      return (Lam repId repBody)
    replace env (Let (NonRec id body) expr) = do
      repBody <- replace env body
      repId <- replaceIdTypes typeToReplace id
      repExpr <- replace (addOneToUniqSet env repId) expr
      return (Let (NonRec repId repBody) repExpr)
    replace env (Let (Rec binds) expr) = do
      let (ids,bodies) = unzip binds
      repIds <- mapM (replaceIdTypes typeToReplace) ids
      let newEnv = addListToUniqSet env repIds
      repBodies <- mapM (replace newEnv) bodies
      repExpr <- replace newEnv expr
      return (Let (Rec (zip repIds repBodies)) repExpr)
    replace env (Case expr id alts) = do
      repExpr <- replace env expr
      repId <- replaceIdTypes typeToReplace id
      let newEnv = addOneToUniqSet env repId
          replaceAlt (con, ids, expr) = do
            repIds <- mapM (replaceIdTypes typeToReplace) ids
            newExpr <- replace (addListToUniqSet newEnv repIds) expr
            return (con, repIds, newExpr)
      repAlts <- mapM replaceAlt alts
      return (Case repExpr repId repAlts)
    replace env (Type ty) = do
      repTy <- (replaceTypeTypes typeToReplace ty)
      return (Type repTy)
    replace env expr = mapExprExprM (replace env) expr -- Con, App

-- replace every occurrence of the given type by a new type variable
-- in an id
replaceIdTypes :: Type -> Id -> CountIO Id
```

```
replaceIdTypes ty = mapVarM (replaceTypeTypes ty)


{-
replace every occurrence of the given type
by a new type variable in a type
the given type is always an algebraic type and may only contain
type variables which are global to the inspected type, i.e. no
locally bound variables
-}
replaceTypeTypes :: Type -> Type -> CountIO Type

replaceTypeTypes _ ty@(TyVarTy tyVar) = return ty
replaceTypeTypes tyToReplace ty@(TyConApp tyCon args)
-- no equivalence modulo alpha conversion needed here
  | ty == tyToReplace = newTITyVar
  | otherwise = do
    repArgs <- mapM (replaceTypeTypes tyToReplace) args
    return (TyConApp tyCon repArgs)
replaceTypeTypes tyToReplace (FunTy arg res) = do
  replaceArg <- replaceTypeTypes tyToReplace arg
  replaceRes <- replaceTypeTypes tyToReplace res
  return (FunTy replaceArg replaceRes)
replaceTypeTypes tyToReplace (ForAllTy var body) = do
  replaceBody <- replaceTypeTypes tyToReplace body
  return (ForAllTy var replaceBody)

{-
Eliminate bound type inference variables in a type, i.e.:
- replace bound type variables by the type they are bound to
-}

elimTypeBounds :: Type -> IO Type

elimTypeBounds tyExpr@(TyVarTy tyVar)
  | isMutTyVar tyVar = do
    content <- readTITyVar tyVar
    case content of
      Nothing -> return tyExpr
      Just ty -> elimTypeBounds ty
  | otherwise = return tyExpr
elimTypeBounds ty = mapTypeTypeM elimTypeBounds ty
  -- note that there are no forall bound type inference variables


{-
Eliminate the type inference variables in an expression, i.e.:
- replace bound type variables by the type they are bound to
- replace FlexiTyVars by TyVars with same Unique
-}

elimExprTyVars :: CoreExpr -> IO CoreExpr
```

```
elimExprTyVars = mapExprTyM elimTypeTyVars

elimTypeTyVars :: Type -> IO Type

elimTypeTyVars ty@(TyVarTy tyVar)
  | isMutTyVar tyVar = do
    content <- readTITyVar tyVar
    case content of
      Nothing -> return (TyVarTy (makeTyVarImmutable tyVar))
      Just ty -> elimTypeTyVars ty
  | otherwise = return ty
elimTypeTyVars (ForAllTy tyVar body) = do
  elimBody <- elimTypeTyVars body
  return $ ForAllTy tyVar elimBody
    -- note that there are no forall bound type inference variables
elimTypeTyVars ty = mapTypeTypeM elimTypeTyVars ty


{-
Eliminate the type inference variables in an expression, i.e.:
- replace bound type variables by the type they are bound to
- any unbound flexiTyVars are bound to a given default type
  this default type may only contain global type variables, i.e,
  no locally bound type variables
-}

elimExprBoundTyVars :: Type -> CoreExpr -> IO CoreExpr

elimExprBoundTyVars defaultTy = mapExprTyM (elimTypeBoundTyVars defaultTy)


elimTypeBoundTyVars :: Type -> Type -> IO Type

elimTypeBoundTyVars defaultTy ty@(TyVarTy tyVar)
  | isMutTyVar tyVar = do
    content <- readTITyVar tyVar
    case content of
      Nothing  -> return defaultTy
      Just ty  -> elimTypeBoundTyVars defaultTy ty
  | otherwise = return ty
elimTypeBoundTyVars defaultTy (ForAllTy tyVar body) = do
  elimBody <- elimTypeBoundTyVars defaultTy body
  return $ ForAllTy tyVar elimBody
    -- note that there are no forall bound type inference variables
elimTypeBoundTyVars defaultTy ty =
  mapTypeTypeM (elimTypeBoundTyVars defaultTy) ty



{-
replace every occurrence of a data constructor of the type to be removed
(e.g. type [Bool], replace Cons Bool, Nil Bool) by
a respective term variable c
```

```
whose type coincides with that of the data constructor, except that all
occurrences of the type to be removed are replaced by a new type variable
(e.g. c : Bool -> beta -> beta, respectively n : beta

Expects that type arguments are always in front of term arguments in
a sequence of applications.
-}

replaceConstructors ::
  ConsToPlaceholders -> [Type] -> CoreExpr -> CountIO CoreExpr

replaceConstructors consToPlaceholders tyArgs e = replace e
  where
  -- replace :: CoreExpr -> IO (CoreExpr)
  replace expr@(App _ _) =
    case collectArgs expr of
      (fun@(Var id), args) -> replaceDefault (getUnique id) args (mkApps fun)
      (fun, args) -> do
        repFun <- replace fun
        repArgs <- mapM replace args
        return (mkApps repFun repArgs)
  replace expr@(Var id) | null tyArgs =
    replaceDefault (getUnique id) [] (mkApps expr)
  replace expr@(Con dataCon args) = replaceDefault
    (getUnique dataCon) args (Con dataCon)
  replace expr = mapExprExprM replace expr

  replaceDefault :: Unique -> [CoreExpr] -> ([CoreExpr] -> CoreExpr)
                     -> CountIO (CoreExpr)
  replaceDefault unique args defaultFun = do
    repArgs <- mapM replace args
    fromMaybe (return $ defaultFun repArgs) $ do
      createPlaceholder <-  consToPlaceholders unique
      remainingRepArgs <- compareTys repArgs tyArgs
      return $ do
        tyVar <- newTITyVar
        return $ mkApps (createPlaceholder tyVar) remainingRepArgs

compareTys :: [CoreExpr] -> [Type] -> Maybe [CoreExpr]

compareTys exprs [] = Just exprs
compareTys (Type tyE : exprs) (ty:tys) | tyE == ty = compareTys exprs tys
compareTys _ _ = Nothing

{-
Reinsert constructors
Replace the constructor placeholders by data constructors,
if their type is not of the desired polymorphic form,
i.e. the type doesn't contain the buildTyVar,
like e.g. c : tau -> buildTyVar -> buildTyVar, n : buildTyVar
(note that we don't care about other type variables in the types
of the constructor placeholders)
-}
```

```
reInsertConstructors :: PlaceholdersToCons -> CoreExpr -> IO CoreExpr

reInsertConstructors placeholdersToCons = insert
  where
  insert expr@(Var id) = do
    maybeCons <- placeholdersToCons id
    return (fromMaybe expr maybeCons)
  insert expr = mapExprExprM insert expr
{-
Instantiate polymorphic functions
Threads bindings of inlineable functions (i.e. the created instances),
bindings for the original definitions to be able to undo instantiation
and a UniqSupply for creating new names.
-}

instantiate :: InlineEnv -> Type -> CoreExpr -> InM (InstEnv, CoreExpr)

instantiate inlineEnv resultTy expr = do
  (_, oldInstEnv, _) <- updateState (\(inInst,_,us)->(inInst,emptyUFM,us))
  instExpr <- go oldInstEnv expr
  (_, newInstEnv, _) <- updateState (\(inInst,_,us)->(inInst,oldInstEnv,us))
  return (newInstEnv, instExpr)
  where
  go :: InstEnv -> CoreExpr -> InM CoreExpr
  go oldInstEnv expr@(App _ (Type _)) =
    case fun of
      Var id -> instantiateTyApp inlineEnv oldInstEnv resultTy expr id tyArgs
      _      -> do
                  funInst <- go oldInstEnv fun
                  return $ mkApps funInst tyArgs
    where
    (fun, tyArgs) = collectArgs expr
  go oldInstEnv expr@(Var id) =
    instantiateTyApp inlineEnv oldInstEnv resultTy expr id []
  go oldInstEnv expr =
    mapExprExprM (go oldInstEnv) expr


instantiateTyApp :: InlineEnv -> InstEnv -> Type
                    -> CoreExpr -> Id -> [CoreArg] -> InM CoreExpr

instantiateTyApp inlineEnv oldInstEnv resultTy expr id tyArgs = do
  (inInst, newInstEnv, _) <- readState
  case do
    instId <- lookupUFM inInst id
     -- same type arguments if any:
    guard $ cheapEqExpr expr (((\(e,_,_)->e) $ fromJust $
      lookupUFM oldInstEnv instId `mplus` lookupUFM newInstEnv instId))
    return $ Var instId
   of
    Just instExpr -> return instExpr
    Nothing ->
```

```
      case lookupUFM inlineEnv id of
        Nothing -> return expr -- unchanged
        Just body ->
          let idResultTy = resultTyOfTy (idType id) in
            if isTyConInTy resultTyCon idResultTy &&
              (isTyInTy resultTy $ resultTyOfTy $ coreExprType expr) then
              -- instantiate the partial application
              instantiateId id expr (tyBetaReduce body tyArgs)
            else if isTyInTy resultTy idResultTy then
              -- instantiate id, without possible type arguments
              instantiateId id (Var id) body
            else return expr -- unchanged
  where
  TyConApp resultTyCon _ = resultTy

{-
Type beta reduction
e.g. (\a b -> e) Int Char ~> e [Int/a] [Char/b]
-}

tyBetaReduce :: CoreExpr -> [CoreArg]  -> CoreExpr
tyBetaReduce expr args = go expr args []
  where
  go :: CoreExpr -> [CoreArg] -> [(TyVar, Type)] -> CoreExpr
  go (Lam tyVar body) (Type ty : args) subs =
    go body args ((tyVar, ty) : subs)
  go expr [] subs =  substExpr (listToUFM subs) expr


substExpr :: UniqFM Type -> CoreExpr -> CoreExpr
substExpr subst = mapExprTy (substTy subst)

substTy :: UniqFM Type -> Type -> Type
substTy subst = mapTypeTyVar $
  \tyVar -> fromMaybe (TyVarTy tyVar) (lookupUFM subst tyVar)

{-
Analysis functions for types
-}

resultTyOfTy :: Type -> Type
resultTyOfTy = lastFunResultTy . snd . splitForAllTys

lastFunResultTy :: Type -> Type
lastFunResultTy (FunTy arg res) = lastFunResultTy res
lastFunResultTy ty = ty

-- searches only for fully applied algebraic type
isTyConInTy :: TyCon -> Type -> Bool
isTyConInTy _ (TyVarTy _) = False
isTyConInTy searchTyCon (TyConApp tyCon tys) =
  searchTyCon == tyCon || any (isTyConInTy searchTyCon) tys
isTyConInTy searchTyCon (FunTy arg res) =
```

```
  isTyConInTy searchTyCon arg || isTyConInTy searchTyCon res
isTyConInTy searchTyCon (ForAllTy _ body) = isTyConInTy searchTyCon body


-- searches only for algebraic type
isTyInTy :: Type -> Type -> Bool
isTyInTy _ (TyVarTy _) = False
isTyInTy searchTy ty@(TyConApp tyCon tys) =
  searchTy == ty || any (isTyInTy searchTy) tys
isTyInTy searchTy (FunTy arg res) =
  isTyInTy searchTy arg || isTyInTy searchTy res
isTyInTy searchTy (ForAllTy _ body) = isTyInTy searchTy body


{-
Instantiation
replace expression by new id
store information for inlining and undoing instantiation
-}

instantiateId :: Id -> CoreExpr -> CoreExpr -> InM CoreExpr

instantiateId oldId oldExpr body = do
  (inInst, instEnv, uniqSupply) <- readState
  let (uniqSupply1, uniqSupply') = splitUniqSupply uniqSupply
      (uniqSupply2, uniqSupply3) = splitUniqSupply uniqSupply'
      newId = mkInstId uniqSupply1 oldId (coreExprType oldExpr)
      renamedBody = rename uniqSupply2 body
  writeState
    ( addToUFM inInst oldId newId
    , addToUFM instEnv newId (oldExpr, renamedBody, newId)
    , uniqSupply3)
  return $ Var newId


mkInstId :: UniqSupply -> Id -> Type -> Id

mkInstId us id ty =
  mkVanillaId (setNameUnique (idName id) (uniqFromSupply us)) ty


mkCopyVar :: Var -> State UniqSupply Var

mkCopyVar var = do
  us <- readState
  let (us1,us2) = splitUniqSupply us
  writeState us2
  return $ setVarUnique var (uniqFromSupply us1)


{-
Rename all locally bound vars of the expression using new uniques
-}
```

```
rename :: UniqSupply -> CoreExpr -> CoreExpr
rename us expr = snd $ deTriv $ flip runStateMT us $ renameExpr emptyUFM expr


renameExpr :: UniqFM IdOrTyVar -> CoreExpr -> State UniqSupply CoreExpr

renameExpr env (Var id) =
  case lookupUFM env id of
    Just reId -> return $ Var reId -- ASK Simon: loss of attached information?
    Nothing -> do
                 reTy <- renameTy env $ idType id
                 return $ Var $ setIdType id reTy
renameExpr env (Lam id body) = do
  reId <- renameId env id
  reBody <- renameExpr (addToUFM env id reId) body
  return $ Lam reId reBody
renameExpr env (Let (NonRec id body) expr) = do
  reId <- renameId env id
  reBody <- renameExpr env body
  reExpr <- renameExpr (addToUFM env id reId) expr
  return $ Let (NonRec reId reBody) reExpr
renameExpr env (Let (Rec binds) expr) = do
  reIds <- mapM (renameId env) ids
  let newEnv = addListToUFM env (zip ids reIds)
  reBodies <- mapM (renameExpr newEnv) bodies
  reExpr <- renameExpr newEnv expr
  return $ Let (Rec (zip reIds reBodies)) reExpr
  where
  (ids,bodies) = unzip binds
renameExpr env (Case scrutinee id alts) = do
  reScrutinee <- renameExpr env scrutinee
  reId <- renameId env id
  reAlts <- mapM (renameAlt (addToUFM env id reId)) alts
  return $ Case reScrutinee reId reAlts
  where
  renameAlt env (con, ids, expr) = do
    reIds <- mapM (renameId env) ids
    reExpr <- renameExpr (addListToUFM env (zip ids reIds)) expr
    return (con, reIds, reExpr)
renameExpr env (Type ty) = do
  reTy <- renameTy env ty
  return $ Type reTy
renameExpr env expr = mapExprExprM (renameExpr env) expr -- Con, App


renameTy :: UniqFM IdOrTyVar -> Type -> State UniqSupply Type

renameTy env ty@(TyVarTy tyVar) = return $
  case lookupUFM env tyVar of
    Just reTyVar -> TyVarTy reTyVar -- ASK Simon: loss of attached information?
    Nothing -> ty
renameTy env (ForAllTy tyVar ty) = do
```

```
  reTyVar <- mkCopyVar tyVar
  reTy <- renameTy (addToUFM env tyVar reTyVar) ty
  return $ ForAllTy reTyVar reTy
renameTy env ty = mapTypeTypeM (renameTy env) ty -- TyConApp, FunTy


renameId :: UniqFM IdOrTyVar -> Id -> State UniqSupply Id

renameId env id = do
  reTy <- renameTy env $ idType id
  mkCopyVar $ setIdType id reTy


{-
uninstantiate
undo instantiations that were not inlined
-}

uninstantiate :: InstEnv -> TyVar -> CoreExpr -> CoreExpr

uninstantiate instEnv resultTyVar expr@(Var id) =
  fromMaybe expr $ do
    (oldExpr, _, _) <- lookupUFM instEnv id
    guard $ not $ isTyVarInTy resultTyVar (idType id)
    return oldExpr
uninstantiate instEnv resultTyVar expr =
  mapExprExpr (uninstantiate instEnv resultTyVar) expr


-- searches only for type variable
isTyVarInTy :: TyVar -> Type -> Bool
isTyVarInTy searchTyVar (TyVarTy tyVar) = searchTyVar == tyVar
isTyVarInTy searchTyVar (TyConApp tyCon tys) =
  any (isTyVarInTy searchTyVar) tys
isTyVarInTy searchTyVar (FunTy arg res) =
  isTyVarInTy searchTyVar arg || isTyVarInTy searchTyVar res
isTyVarInTy searchTyVar (ForAllTy _ body) = isTyVarInTy searchTyVar body


--------------------------------------------------------------------------
-- End
```

A.12 Core.hs

```
-- syntax for F expressions
-- By Olaf Chitil, with minor changes by Kirsten Chevalier
```

```
module Core where

import Prelude -- hiding (Rec)
import ThisUnique
import IOExts (IORef, newIORef, readIORef, writeIORef)
import PrettyCore
import ThisUtil

data Unused = Unused

--------------------- data and type constructors ----------------------
data TyCon = List | Tuple | Bool | Int | Char | Kind deriving (Eq, Show)
data Con = C Name deriving Eq

cons = C Named{n_occ="Cons", n_uniq=consDataConKey}
nil = C Named{n_occ="Nil", n_uniq=nilDataConKey}
tuple = C Named{n_occ="Tuple", n_uniq= -4}
true = C Named{n_occ="True", n_uniq= -5}
false = C Named{n_occ="False", n_uniq= -6}

patType :: Pat -> Type
patType pat = case pat of
               ConPat c  -> conType c
               LitPat (MachChar _) -> TyConApp Char []
-- this actually should be something more general
               DEFAULT -> (TyConApp List [(TyConApp Int [])])

-- determine the type of a constant, from Const.lhs
conType :: Con -> Type
conType = dataConType

-- determine type of a data constructor, signature from DataCon.lhs
dataConType :: Con -> Type

dataConType (C Named{n_occ="Cons"}) =
  ForAllTy alpha $ FunTy alphaTy
                   (FunTy (TyConApp List [alphaTy]) (TyConApp List [alphaTy]))
dataConType (C Named{n_occ="Nil"})   = ForAllTy alpha $ TyConApp List [alphaTy]
dataConType (C Named{n_occ="Tuple"}) =
  ForAllTy alpha $ ForAllTy beta $
    alphaTy `FunTy` betaTy `FunTy` TyConApp Tuple [alphaTy,betaTy]
dataConType (C Named{n_occ="True"})  = TyConApp Bool []
dataConType (C Named{n_occ="False"}) = TyConApp Bool []
dataConType (C Named{n_occ="C#"}) = TyConApp Char []
dataConType (C Named{n_occ="I#"}) = TyConApp Int []

alpha, beta, gamma, delta :: TyVar
alpha = mkTyVar Named{n_occ="a",n_uniq= -10} starKind
beta = mkTyVar Named{n_occ="b",n_uniq= -11} starKind
gamma = mkTyVar Named{n_occ="c",n_uniq= -12} starKind
delta = mkTyVar Named{n_occ="d",n_uniq= -13} starKind

alphaTy = TyVarTy alpha
```

```
betaTy = TyVarTy beta
gammaTy = TyVarTy gamma
deltaTy = TyVarTy delta

charTy = TyConApp Char []
intTy = TyConApp Int []
boolTy = TyConApp Bool []

starKind :: Type
starKind = TyConApp Kind []

instance Uniquable Con where
  getUnique (C name) = n_uniq name

-------------------------------------------------------------------------
-- module SrcLoc

data SrcLoc
  = NoSrcLoc
  | SrcLoc String    -- A precise location (file name)
  | UnhelpfulSrcLoc String

mkGeneratedSrcLoc   = UnhelpfulSrcLoc "<compiler-generated-code>"
-------------------------------------------------------------------------
-- module OccName

type OccName = String  -- extremly simplified

-------------------------------------------------------------------------
-- module Name

data Name = Named {n_occ :: OccName, n_uniq :: Unique}

nameUnique :: Name -> Unique
nameUnique = n_uniq

mkLocalName :: Unique -> OccName -> SrcLoc -> Name
mkLocalName uniq occ loc = Named { n_uniq = uniq, n_occ = occ }

mkSysLocalName :: Unique -> String -> Name
mkSysLocalName uniq fs = Named { n_uniq = uniq,
                                n_occ = fs}

mkDerivedName :: (OccName -> OccName)
              -> Name           -- Base name
              -> Unique         -- New n_uniq
              -> Name           -- Result is always a value name

mkDerivedName f name uniq = name {n_uniq = uniq, n_occ = f (n_occ name)}

-- When we renumber/rename things, we need to be
-- able to change a Name's Unique to match the cached
-- one in the thing it's the name of.  If you know what I mean.
```

```
setNameUnique name uniq = name {n_uniq = uniq}

instance Uniquable Name where
    getUnique = nameUnique

instance Eq Name where
  n1 == n2 = nameUnique n1 == nameUnique n2

--------------------------------------------------------------------------
-- module Var

data Var
  = VarG {
        varName    :: Name,
        realUnique :: Unique, -- Key for fast comparison -- Need to convert
                              -- from Int#
        varType    :: Type,
        varDetails :: VarDetails
     }

data VarDetails
  = AnId
  | TyVar
  | MutTyVar (IORef (Maybe Type))       -- Used during unification;
             Bool                       -- True <=> this is a type signature
                                        -- variable, which should not be unified
                                        -- with a non-tyvar type
  | UVar                                -- Usage variable
  | Build Type

instance Uniquable Var where
  getUnique = realUnique

instance Eq Var where
  var1 == var2 = realUnique var1 == realUnique var2


varUnique :: Var -> Unique
varUnique (VarG {realUnique = uniq}) = uniq

setVarUnique :: Var -> Unique -> Var
setVarUnique var uniq = var {realUnique = uniq,
                             varName = setNameUnique (varName var) uniq}

setVarName :: Var -> Name -> Var
setVarName var new_name
  = var { realUnique = getUnique new_name, varName = new_name }

setVarType :: Var -> Type -> Var
setVarType var ty = var {varType = ty}

---------------------- type variables: -------------------------------
```

```
type TyVar = Var

tyVarName = varName
tyVarKind = varType

setTyVarUnique = setVarUnique
setTyVarName   = setVarName

mkTyVar :: Name -> Kind -> TyVar
mkTyVar name kind = VarG { varName    = name
                        , realUnique = nameUnique name
                        , varType    = kind
                        , varDetails = TyVar
                        }

mkSysTyVar :: Unique -> Kind -> TyVar
mkSysTyVar uniq kind = VarG { varName    = name
                           , realUnique = uniq
                           , varType    = kind
                           , varDetails = TyVar}
                     where
                       name = mkSysLocalName uniq "t"


newMutTyVar :: Name -> Kind -> IO TyVar
newMutTyVar name kind =
  do loc <- newIORef Nothing
     return (VarG { varName = name,
                   realUnique = nameUnique name,
                   varType = kind,
                   varDetails = MutTyVar loc False})

readMutTyVar :: TyVar -> IO (Maybe Type)
readMutTyVar (VarG {varDetails = MutTyVar loc _}) = readIORef loc

writeMutTyVar :: TyVar -> Maybe Type -> IO ()
writeMutTyVar (VarG {varDetails = MutTyVar loc _}) val = writeIORef loc val

makeTyVarImmutable :: TyVar -> TyVar
makeTyVarImmutable tyvar = tyvar { varDetails = TyVar}

isTyVar :: Var -> Bool
isTyVar (VarG {varDetails = details}) = case details of
                                        TyVar        -> True
                                        MutTyVar _ _ -> True
                                        other        -> False

isMutTyVar :: Var -> Bool
isMutTyVar (VarG {varDetails = MutTyVar _ _}) = True
isMutTyVar other                              = False

-------------------   id construction: ---------------------------
```

```
type Id = Var
type IdOrTyVar = Var

idName    = varName
idType    = varType
idUnique  = varUnique

setIdUnique :: Id -> Unique -> Id
setIdUnique = setVarUnique

setIdName :: Id -> Name -> Id
setIdName = setVarName

mkIdVar :: Name -> Type -> Id
mkIdVar name ty
  = VarG {varName = name, realUnique = nameUnique name, varType = ty,
         varDetails = AnId }

isId :: Var -> Bool
isId (VarG {varDetails = AnId}) = True
isId other                      = False

-------------------------------------------------------------------------
-- basicTypes/Id

mkId :: Name -> Type -> Id
mkId = mkIdVar

mkVanillaId :: Name -> Type -> Id
mkVanillaId name ty = mkId name ty {-vanillaIdInfo-}

-- SysLocal: for an Id being created by the compiler out of thin air...
-- UserLocal: an Id with a name the user might recognize...
mkUserLocal :: OccName -> Unique -> Type -> SrcLoc -> Id
mkSysLocal  :: String -> Unique -> Type -> Id

mkSysLocal  fs uniq ty      = mkVanillaId (mkSysLocalName uniq fs)      ty
mkUserLocal occ uniq ty loc = mkVanillaId (mkLocalName    uniq occ loc) ty

setIdType :: Id -> Type -> Id
setIdType id ty = setVarType id ty


-------------------------------------------------------------------------
-- types/Type

-- data types

type SuperKind = Type
type Kind      = Type

data Type
  = TyVarTy TyVar
```

```
  | TyConApp TyCon [Type]  -- Might not be saturated.
  | FunTy Type Type
  | ForAllTy TyVar Type    -- TypeKind
  deriving Eq -- Sketchy!

instance Show Type where
 show t = pretty t

infixr 5 `FunTy`

-- constructor specific functions:

mkTyVarTy  :: TyVar   -> Type
mkTyVarTy  = TyVarTy

mkFunTy :: Type -> Type -> Type
mkFunTy arg res = FunTy arg res

mkFunTys :: [Type] -> Type -> Type
mkFunTys tys ty = foldr FunTy ty tys

splitFunTy_maybe :: Type -> Maybe (Type, Type)
splitFunTy_maybe (FunTy arg res) = Just (arg, res)
splitFunTy_maybe other           = Nothing

funResultTy :: Type -> Type
funResultTy (FunTy arg res) = res
funResultTy ty              = error "funResultTy"

mkTyConApp :: TyCon -> [Type] -> Type
mkTyConApp = TyConApp

isForAllTy :: Type -> Bool
isForAllTy (ForAllTy tyvar ty) = True
isForAllTy _                = False

splitForAllTys :: Type -> ([TyVar], Type)
splitForAllTys ty = split ty ty []
   where
     split orig_ty (ForAllTy tv ty) tvs = split ty ty (tv:tvs)
     split orig_ty t                tvs = (reverse tvs, orig_ty)
---------------------------------------------------------------------------
-- CoreSyn

-- the main data type

infixl 8 `App`  -- App brackets to the left

data Expr b     -- "b" for the type of binders,
  = Var   Id
  | Lit   Literal                --NEW
  | Con   Con [Arg b]         -- Guaranteed saturated -- Not in Core
  | App   (Expr b) (Arg b)
```

```
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b [Alt b]  -- Binder gets bound to value of scrutinee
                              -- DEFAULT case must be last, if it occurs at all
  | Type  Type       -- This should only show up at the top level of an Arg

------- Literal ---------------
data Literal = MachChar Char | MachInt Integer

litType (MachChar c) = (TyConApp Char [])
litType (MachInt i) = (TyConApp Int [])
-------------------------------

type Arg b = Expr b         -- Can be a Type

type Alt b = (Pat, [b], Expr b)

        -- (DEFAULT, [], rhs) is the default alternative
        -- Remember, a Con can be a literal or a data constructor
data Pat = ConPat Con | LitPat Literal | DEFAULT

data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]

type CoreBndr = IdOrTyVar
type CoreExpr = Expr CoreBndr
type CoreArg  = Arg  CoreBndr
type CoreBind = Bind CoreBndr
type CoreAlt  = Alt  CoreBndr

mkApps :: Expr b -> [Arg b] -> Expr b
mkApps f args = foldl App f args

collectArgs :: Expr b -> (Expr b, [Arg b])
collectArgs expr
  = go expr []
  where
    go (App f a) as = go f (a:as)
    go e         as = (e, as)

isTypeArg (Type _) = True
isTypeArg other    = False

applyTy :: Type -> Type -> Type
applyTy (ForAllTy tv ty) arg = substTy' ([(tv,arg)]) ty
applyTy other arg = error("applyTy: trying to apply a non-forall-type,  namely "
  ++ (prettyTy other))

applyTys :: Type -> [Type] -> Type
applyTys fun_ty arg_tys
 = go [] fun_ty arg_tys
 where
   go env ty               []         = substTy' env ty
```

```
   go env (ForAllTy tv ty) (arg:args) = go ((tv,arg):env) ty args
   go _ other _ = error("applyTys: trying to apply a non-forall-type, namely "
++ (pretty other))

substTy' :: [(TyVar,Type)] -> Type -> Type
substTy' env = mapTypeTyVar (\var -> case lookup var env of
  Just ty -> ty
  Nothing -> TyVarTy var)

--------------------------------------------------------------------------
-- module CoreUtils

coreExprType :: CoreExpr -> Type

coreExprType (Var var)              = idType var
coreExprType e@(App _ _)
  = case collectArgs e of
        (fun, args) -> applyTypeToArgs e (coreExprType fun) args
coreExprType _ = error "coreExprType only partially defined"

-- The first argument is just for debugging
applyTypeToArgs :: CoreExpr -> Type -> [CoreExpr] -> Type
applyTypeToArgs e op_ty [] = op_ty

applyTypeToArgs e op_ty (Type ty : args)
  = -- Accumulate type arguments so we can instantiate all at once
    applyTypeToArgs e (applyTys op_ty tys) rest_args
  where
    (tys, rest_args)        = go [ty] args
    go tys (Type ty : args) = go (ty:tys) args
    go tys rest_args        = (reverse tys, rest_args)

{-
@cheapEqExpr@ is a cheap equality test which bails out fast!
        True  => definitely equal
        False => may or may not be equal
-}

cheapEqExpr :: Expr b -> Expr b -> Bool

cheapEqExpr (Var v1) (Var v2) = v1==v2
cheapEqExpr (Con con1 args1) (Con con2 args2)
  = con1 == con2 &&
    and (zipWith cheapEqExpr args1 args2)

cheapEqExpr (App f1 a1) (App f2 a2)
  = f1 `cheapEqExpr` f2 && a1 `cheapEqExpr` a2

cheapEqExpr (Type t1) (Type t2) = t1 == t2

cheapEqExpr _ _ = False
```

```
smpId text un ty = mkVanillaId (Named text un) ty
smpTyVar text un = mkTyVar Named{n_occ=text,n_uniq=un} starKind

nameEquals var string = ((n_occ (varName var)) == string)
```

A.13 ThisUtil.hs

```
-- miscellaneous utilities
-- By Olaf Chitil
module ThisUtil where
import Core

fst3 (x,_,_) = x
snd3 (_,x,_) = x
thrd3 (_,_,x) = x

-- map a monadic function on type component of a Var
mapVarM :: Monad m => (Type -> m Type) -> Var -> m Var
mapVarM f id@VarG{varType = ty} = do
  mapTy <- f ty
  return (id{varType = mapTy})

-- literals always have the same type
mapLitM :: Monad m => (Type -> m Type) -> Literal -> m Literal
mapLitM f x = return(x)

-- map a function on type component of a Var
mapVar :: (Type -> Type) -> Var -> Var
mapVar f id@VarG{varType = ty} = id{varType = f ty}

{-
Map a monadic function on all type components of a type
-}

mapTypeTypeM :: Monad m => (Type -> m Type) -> (Type -> m Type)

mapTypeTypeM f ty@(TyVarTy _) = return ty
mapTypeTypeM f (TyConApp con tys) = do
  mapTys <- mapM f tys
  return $ TyConApp con mapTys
mapTypeTypeM f (FunTy fun arg) = do
  mapFun <- f fun
  mapArg <- f arg
  return $ FunTy mapFun mapArg
mapTypeTypeM f (ForAllTy tyVar body) = do
```

```
  mapBody <- f body
  return $ ForAllTy tyVar mapBody

{-
map a function on all type components of an expression
-}

mapExprTy :: (Type -> Type) -> (CoreExpr -> CoreExpr)

mapExprTy f (Var id) = Var (mapVar f id)
mapExprTy f (Lam id body) = Lam (mapVar f id) (mapExprTy f body)
mapExprTy f (Let (NonRec id body) expr) =
  Let (NonRec (mapVar f id) (mapExprTy f body)) (mapExprTy f expr)
mapExprTy f (Let (Rec binds) expr) =
  Let (Rec (map mapBind binds)) (mapExprTy f expr)
  where
  mapBind (id, body) = (mapVar f id, mapExprTy f body)
mapExprTy f (Case scrutinee id alts) =
  Case (mapExprTy f scrutinee) (mapVar f id) (map mapAlt alts)
  where
  mapAlt (con, ids, expr) = (con, map (mapVar f) ids, mapExprTy f expr)
mapExprTy f (Type ty) = Type (f ty)
mapExprTy f expr = mapExprExpr (mapExprTy f) expr -- Con, Fun


{-
map a function on all expression components of an expression
-}

mapExprExpr :: (CoreExpr -> CoreExpr) -> (CoreExpr -> CoreExpr)

mapExprExpr f (Con con args) = Con con (map f args)
mapExprExpr f (App fun arg) = App (f fun) (f arg)
mapExprExpr f (Lam id body) = Lam id (f body)
mapExprExpr f (Let (NonRec id body) expr) =
  Let (NonRec id (f body)) (f expr)
mapExprExpr f (Let (Rec binds) expr) =
  Let (Rec (map mapBind binds)) (f expr)
  where
  mapBind (id, body) = (id, f body)
mapExprExpr f (Case scrutinee id alts) =
  Case (f scrutinee) id (map mapAlt alts)
  where
  mapAlt (con, ids, expr) = (con, ids, f expr)
mapExprExpr f expr = expr -- Var, Type


{-
map a monadic function on all type components of an expression
-}

mapExprTyM :: Monad m => (Type -> m Type) -> (CoreExpr -> m CoreExpr)
```

```
mapExprTyM f (Lit l) = do
  mapId <- mapLitM f l
  return(Lit mapId)
mapExprTyM f (Var id) = do
  mapId <- mapVarM f id
  return (Var mapId)
mapExprTyM f (Con con args) = do
  elimArgs <- mapM (mapExprTyM f) args
  return (Con con elimArgs)
mapExprTyM f (App fun arg) = do
  elimFun <- mapExprTyM f fun
  elimArg <- mapExprTyM f arg
  return (App elimFun elimArg)
mapExprTyM f (Lam id body) = do
  mapId <- mapVarM f id
  mapBody <- mapExprTyM f body
  return (Lam mapId mapBody)
mapExprTyM f (Let (NonRec id body) expr) = do
  mapId <- mapVarM f id
  mapBody <- mapExprTyM f body
  mapExpr <- mapExprTyM f expr
  return (Let (NonRec mapId mapBody) mapExpr)
mapExprTyM f (Let (Rec defs) expr) = do
  mapDefs <- mapM mapBindingsMf defs
  mapExpr <- mapExprTyM f expr
  return (Let (Rec mapDefs) mapExpr)
  where
  mapBindingsMf (id, expr) = do
    mapId <- mapVarM f id
    mapExpr <- mapExprTyM f expr
    return (mapId, mapExpr)
mapExprTyM f (Case expr id alts) = do
  mapExpr <- mapExprTyM f expr
  mapId <- mapVarM f id
  mapAlts <- mapM mapAltMf alts
  return (Case mapExpr mapId mapAlts)
  where
  mapAltMf (con, ids, expr) = do
    mapIds <- mapM (mapVarM f) ids
    mapExpr <- mapExprTyM f expr
    return (con, mapIds, mapExpr)
mapExprTyM f (Type ty) = do
  mapTy <- f ty
  return (Type mapTy)

{-
map a monadic function on all expression components of an expression
-}

mapExprExprM :: Monad m => (Expr a -> m (Expr a)) -> Expr a -> m (Expr a)

mapExprExprM f (Con con args) = do
  recArgs <- mapM f args
```

```
  return (Con con recArgs)
mapExprExprM f (App fun arg) = do
  recFun <- f fun
  recArg <- f arg
  return (App recFun recArg)
mapExprExprM f (Lam id body) = do
  recBody <- f body
  return (Lam id recBody)
mapExprExprM f (Let (NonRec id body) expr) = do
  recBody <- f body
  recExpr <- f expr
  return (Let (NonRec id recBody) recExpr)
mapExprExprM f (Let (Rec defs) expr) = do
  let (ids,bodies) = unzip defs
  recBodies <- mapM f bodies
  recExpr <- f expr
  return (Let (Rec (zip ids recBodies)) recExpr)
mapExprExprM f (Case expr id alts) = do
  recExpr <- f expr
  let (cons,idss,exprs) = unzip3 alts
  recExprs <- mapM f exprs
  return (Case recExpr id (zip3 cons idss recExprs))
mapExprExprM f expr = return expr  -- Var, Type

mapExprBoolM :: Monad m => (Expr a -> m (Bool)) -> Expr a -> m (Bool)

mapExprBoolM f (Con con args) = do
  recArgs <- (mapM f args)
  return (and recArgs)
mapExprBoolM f (App fun arg) = do
  recFun <- f fun
  recArg <- f arg
  return (recFun && recArg)
mapExprBoolM f (Lam id body) = do
  recBody <- f body
  return recBody
mapExprBoolM f (Let (NonRec id body) expr) = do
  recBody <- f body
  recExpr <- f expr
  return (recBody && recExpr)
mapExprBoolM f (Let (Rec defs) expr) = do
  let (ids,bodies) = unzip defs
  recBodies <- mapM f bodies
  recExpr <- f expr
  return ((and recBodies) && recExpr)
mapExprBoolM f (Case expr id alts) = do
  recExpr <- f expr
  let (cons,idss,exprs) = unzip3 alts
  recExprs <- mapM f exprs
  return (recExpr && (and recExprs))
mapExprBoolM f exp = do
  res <- f exp
  return res  -- Var, Type
```

```
mapExprBool ::  (Expr a -> Bool) -> Expr a -> Bool

mapExprBool f (Con con args) = (all f args)
mapExprBool f (App fun arg) = ((f fun) && (f arg))
mapExprBool f (Lam id body) = (f body)
mapExprBool f (Let (NonRec id body) expr) = (f body) && (f expr)
mapExprBool f (Let (Rec defs) expr) =
  let (ids,bodies) = unzip defs
  in
  ((all f bodies) && (f expr))
mapExprBool f (Case expr id alts) =
  let (cons,idss,exprs) = unzip3 alts
  in
  ((f expr) && (all f exprs))
mapExprBool f exp = (f exp)


{-
Map a function on the (unbound) type variable components of a type
-}

mapTypeTyVar :: (TyVar -> Type) -> (Type -> Type)

mapTypeTyVar f (TyVarTy tyVar) = f tyVar
mapTypeTyVar f ty = mapTypeType (mapTypeTyVar f) ty

{-
Map a function on all type components of a type
-}

mapTypeType :: (Type -> Type ) -> (Type -> Type)

mapTypeType f ty@(TyVarTy tyVar) = ty
mapTypeType f (TyConApp tyCon args) = TyConApp tyCon (map f args)
mapTypeType f (FunTy arg res) = FunTy (f arg) (f res)
mapTypeType f (ForAllTy tyVar body) = ForAllTy tyVar (f body)
```

A.14 ThisUnique.hs

```
{-
Uniques
UniqSupply
Containers for Uniques: UniqFM, UniqSet
-- By Olaf Chitil
-}
module ThisUnique where

-- from Unique.lhs
class Uniquable a where
    getUnique :: a -> Unique
```

```
type Unique = Integer

instance Uniquable Integer where
 getUnique u = u

consDataConKey :: Unique
nilDataConKey :: Unique
listTyConKey :: Unique
consDataConKey = -1
nilDataConKey = -2
listTyConKey = -3

-- UniqSupply

newtype UniqSupply = US Integer

mkMyUniqSupply :: Integer -> UniqSupply
splitUniqSupply :: UniqSupply -> (UniqSupply, UniqSupply)
uniqFromSupply  :: UniqSupply -> Unique
uniqsFromSupply :: Int -> UniqSupply -> [Unique]

mkMyUniqSupply u = US u
splitUniqSupply (US u) = (US (2*u), US (2*u+1))
uniqFromSupply (US u) = u
uniqsFromSupply 0 _ = []
uniqsFromSupply (n+1) us =
  uniqFromSupply us : uniqsFromSupply n (fst $ splitUniqSupply us)

newtype UniqFM elt = UFM [(Unique,elt)] deriving Show
emptyUFM        :: UniqFM elt
addToUFM        :: Uniquable key => UniqFM elt -> key -> elt  -> UniqFM elt
listToUFM       :: Uniquable key => [(key,elt)] -> UniqFM elt
listToUFM_Directly
                :: [(Unique, elt)] -> UniqFM elt
addListToUFM    :: Uniquable key => UniqFM elt -> [(key,elt)] -> UniqFM elt
lookupUFM       :: Uniquable key => UniqFM elt -> key -> Maybe elt
lookupUFM_Directly :: UniqFM elt -> Unique -> Maybe elt
foldUFM         :: (elt -> a -> a) -> a -> UniqFM elt -> a
eltsUFM         :: UniqFM elt -> [elt]
ufmToList       :: UniqFM elt -> [(Unique, elt)]

emptyUFM = UFM []
addToUFM (UFM xs) key elt = UFM ((getUnique key,elt):xs)
listToUFM keyElts = addListToUFM emptyUFM keyElts
listToUFM_Directly = UFM
addListToUFM (UFM xs) keyElts = UFM ((map (\(k,e) ->
      (getUnique k,e)) keyElts) ++ xs)
lookupUFM (UFM xs) key = lookup (getUnique key) xs
lookupUFM_Directly (UFM xs) u = lookup u xs
foldUFM c n ufm = foldr c n $ eltsUFM ufm
eltsUFM (UFM xs) = snd $ unzip xs
ufmToList (UFM xs) = xs
```

```
newtype UniqSet a = MkUniqSet (UniqFM a) deriving Show
emptyUniqSet :: UniqSet a
emptyUniqSet = MkUniqSet emptyUFM
addOneToUniqSet :: Uniquable a => UniqSet a -> a -> UniqSet a
addOneToUniqSet (MkUniqSet set) x = MkUniqSet (addToUFM set x x)
addListToUniqSet :: Uniquable a => UniqSet a -> [a] -> UniqSet a
addListToUniqSet (MkUniqSet set) xs = MkUniqSet (addListToUFM set [(x,x) | x<-xs])
lookupUniqSet :: Uniquable a => UniqSet a -> a -> Maybe a
lookupUniqSet (MkUniqSet set) x = lookupUFM set x
foldUniqSet :: (a -> b -> b) -> b -> UniqSet a -> b
foldUniqSet c n (MkUniqSet set) = foldUFM c n set

uniqSetToList :: UniqSet a -> [a]
uniqSetToList = foldUniqSet (:) []

lookupUniqSetUniq :: Uniquable a => UniqSet elt -> a -> Maybe elt
lookupUniqSetUniq (MkUniqSet set) key = lookupUFM set key
```

A.15 MonadTransformer.hs

```
{-
Monad transformers
State monad transformer
Maybe monad transformer
By Olaf Chitil
-}

module MonadTransformer where

 import Monad(MonadPlus(..))


 class MonadT t where
   lift :: Monad m => m a -> t m a
   liftInside :: (Monad m, MonadT t2) => t m a -> t (t2 m) a


    {-
     The trivial Monad
    -}

 newtype Triv a = Triv a

 instance Monad Triv where
     return = Triv
     (Triv x) >>= f = f x

 deTriv :: Triv a -> a
 deTriv (Triv x) = x
```

```
  {-
  The state monad transformer
  -}

newtype StateT s m a = STM (s -> m (s,a))

instance Monad m => Monad (StateT s m) where
    return x = STM (\s -> return (s,x))
    (STM f) >>= gsm = STM (\s1 -> do -- >>= is the same as "bindM"
                                    (s2,x) <- f s1
                                -- implicitly use bind of little "m"
                                    let STM g = gsm x
                                    g s2)

updateState :: Monad m => (s -> s) -> StateT s m s
updateState f = STM (\s -> return (f s, s))

readState :: Monad m => StateT s m s
readState = updateState id

writeState :: Monad m => s -> StateT s m s
writeState s = updateState (\_ -> s)

runStateMT :: Monad m => StateT s m a -> s -> m (s,a)
runStateMT (STM f) s = f s


instance MonadT (StateT s) where
  lift m = STM (\s -> do
                      x <- m
                      return (s,x) )
  liftInside (STM f) = STM (\s -> lift (f s))


type State s a = StateT s Triv a


 {-
 The Maybe monad transformer
 -}

newtype MaybeT m a = MT (m (Maybe a))

instance Monad m => Monad (MaybeT m) where
   return x = MT $ return $ Just x
   (MT m1) >>= f = MT $ do
    res <- m1
    case res of
     Just x  -> do
                 let MT m2 = f x
                 m2
     Nothing -> return Nothing
```

```
runMaybeT :: MaybeT m a -> m (Maybe a)
runMaybeT (MT m) = m

instance MonadT MaybeT where
    lift m = MT $ do x <- m; return $ Just x
    liftInside (MT m) = MT (lift m)

instance Monad m => MonadPlus (MaybeT m) where
    mzero = MT $ return $ Nothing
    (MT m1) `mplus` (MT m2) = MT $ do
       res <- m1
       case res of
         Just x  -> return res
         Nothing -> m2
```

A.16 CountIO.hs

```
{-

Idea: (CountIO a) is a Monad like (IO a) except that it supports the operations

  getCounter :: CountIO Int
  incCounter :: CountIO ()
  resetCounter :: CountIO ()

-- By Kirsten Chevalier and Franklyn Turbak
-}

module CountIO where

import MonadTransformer
import Maybe(isJust, fromJust, fromMaybe)

type TIM a = MaybeT IO a

type TIMC a = MaybeT (StateT Integer IO) a -- acts like: CountIO (Maybe a)

tim2timc :: TIM a -> TIMC a
tim2timc (MT m) = (MT (liftCount m))

returnIO :: a -> IO a
returnIO x = return x

io2iomaybe :: (IO a) -> (IO (Maybe a))
io2iomaybe io = do {x <- io;
                  return (Just x)}

io2timc :: (IO a) -> (TIMC a)
io2timc io = MT (liftCount (io2iomaybe io))

type CountIO a = StateT Integer IO a
```

```
getCounter :: CountIO Integer
getCounter = readState

incCounter :: CountIO ()
incCounter = do {updateState (\x -> x+ 1);
                 return ()}

resetCounter :: CountIO ()
resetCounter = do {updateState (\x -> 0);
                   return ()}

runCounter :: (CountIO a)  -> IO a
runCounter cm = do{(finalCounter, result) <- runStateMT cm 0;
                   return result}

runCounterWith :: (CountIO a) -> Integer -> IO (a, Integer)
runCounterWith cm init = (do{(finalCounter, result) <- runStateMT cm init;
                   return (result, finalCounter)})

liftCount :: (IO a) -> (CountIO a)
liftCount = lift

printCounter :: CountIO ()
printCounter = do {count <- getCounter;
                   lift (putStr ("Count = " ++ (show count) ++ "\n"));
                   return ()
                   }

test :: IO ()
test = runCounter (do {printCounter;
       incCounter;
                       printCounter;
       incCounter;
                       printCounter
                       })
```

A.17 PrettyCore.hs

```
---------------------------------------------------------------------------
-- Pretty Printing
-- by Olaf Chitil
module PrettyCore where

import Pretty
import Core

class Ppr a where
  ppr :: Bool -> a -> Doc
  -- the boolean value implements a simple heuristic for
  -- avoiding some unnecessary braces
```

```
class PprT a where
  pprT :: a -> Doc

pprParen :: Bool -> Doc -> Doc
pprParen True  doc = parens doc
pprParen False doc = doc

instance (Ppr b, PprT b) => Ppr (Expr b) where
--  ppr _ (Var x)         = ppr False x
  ppr p (Lit x)         = ppr False x
  ppr p (Var x)         = pprParen p (pprT x)
  ppr _ (Con con [])    = ppr False con
  ppr p (Con con args)  = pprParen p
          $ ppr False con <+> (braces $ sep (map (ppr True) args))
  ppr p expr@(App e1 e2) =
    pprParen p $ ppr False fun <+> sep (map (ppr True) args)
    where
    (fun, args) = collectArgs expr
  ppr p (Lam x e)       = pprParen p
          $ char '\\' <> pprT x <+> text "->" <+> ppr False e
  ppr p (Let bind e)    = pprParen p
          $ text "let" <+> ppr False bind
          $$ (nest 2 $ text "in" <+> ppr False e)
  ppr p (Case e x alts)
    = sep [sep [text "case" <+> ppr False e,
                text "of" <+> ppr True x <+> char '{'],
         nest 6 (sep (punctuate semi (map pprAlt alts))),
         char '}']
  ppr p (Type ty) = ppr p ty

pprAlt :: (Ppr b, PprT b) => Alt b -> Doc
pprAlt (con, xs, e)
  = hang (ppr False con <+> hsep (map (ppr True) xs) <+> text "->") 4
         (ppr False e)

instance (Ppr b, PprT b) => Ppr (Bind b) where
  ppr _ (NonRec x e) = pprEquation x e
  ppr _ (Rec eqs) = braces $ vcat (map (uncurry pprEquation) eqs)

pprEquation :: (PprT x, Ppr e) => x -> e -> Doc
pprEquation x e = sep [pprT x, equals <+> ppr False e <> semi]

instance Ppr Type where
  ppr _ (TyVarTy a) = ppr False a
  ppr _ (TyConApp con []) = ppr False con
  ppr p (TyConApp List [arg]) = text "[" <> ppr False arg <> text "]"
  ppr p (TyConApp Tuple [left,right]) =
    text "(" <> ppr False left <> text "," <> ppr False right <> text ")"
  ppr p (TyConApp con args) =
    pprParen p $ ppr False con <+> hsep (map (ppr True) args)
  ppr p (FunTy t1 t2) = pprParen p $
(ppr True t1 <+> text "->" <+> ppr False t2)
  ppr p (ForAllTy tyVar body) =
```

```
    pprParen p $
     text "forall" <+> ppr False tyVar <> text"." <+> ppr False body

instance Ppr TyCon where
  ppr _ List  = text "[]"
  ppr _ Tuple = text "(,)"
  ppr _ Bool  = text "Bool"
  ppr _ Kind  = text "*"
  ppr _ Int = text "Int"
  ppr _ Char = text "Char"

instance Ppr Con where
  ppr p (C name) = ppr p name

instance Ppr Pat where
  ppr p (ConPat con) = ppr p con
  ppr p (LitPat lit) = ppr p lit
  ppr p DEFAULT = text "DEFAULT"

instance Ppr Var where
  ppr _ var = ppr False (varName var)

instance PprT Var where
  pprT var =  ppr False (varName var) <+> char ':' <+> ppr False (varType var)

instance Ppr Literal where
  ppr _ (MachInt x) = (text (show x))
  ppr _ (MachChar x) = (text (show x))

instance Ppr Name where
-- Right now the text "<> integer u" is commented out so as
-- not to print out Uniques. Uncomment it to print out Uniques.
  ppr _ (Named name u) = text name <> (if (u > 10000) then (text "") else (integer u))

pretty :: Ppr a => a -> String
pretty = render . ppr False

prettyTy :: Type -> String
prettyTy = render . ppr False

prettyId :: Id -> String
prettyId = render . ppr False

prettyIO :: Ppr a => a -> IO ()
prettyIO = putStr . pretty

prettyList :: Ppr a => [a] -> String
prettyList exprs = (concat (map (\ x -> (pretty x) ++ " ") exprs))
```

Appendix B

Running Test Cases

This appendix is intended for someone who might be running and/or modifying the code.

All the code for the type-inference-based deforestation system resides in the directory `/home/lumberjacks/krc/lumberjack/Inference`, which lives on the file server named `crete` and should be accessible from all the Aegean machines. The instructions below assume that this is your current working directory.

Among other things, this directory contains a file called `Makefile`, which contains rules for how to compile the code. When you make a change to the code, you can compile it all at once by typing `make`. This creates an executable program called `all`. Now you can run it on any Haskell program. There are some test programs in the `Test` subdirectory; ones on which type-inference-based deforestation definitely runs are in the `Working` subdirectory of that directory. So, if you wanted to run the program on the working test case called `appendF.hs`, you would type:

```
$ ./all Test/Working/appendF.hs -fhi-version=408
```

The `-fhi-version=408` is necessary, and tells the GHC front-end which version of interface files to expect. (Interface files have names that end in `.hi` – if you use GHC to compile a Haskell program called `foo.hs`, it will create a file called `foo.hi` that tells GHC which definitions `foo.hs` exports. In the `Inference` directory, there are a number of links to interface files, all of whose names begin with `Prel` (for "Prelude"), and these links must be present to run the GHC front-end.)

In the `Test` subdirectory, there is a shell script called `autotest` which runs all the test cases in the `Working` subdirectory. To run the script and redirect the output to the file `results.may29`, you would type:

```
$ Test/autotest >&results.may29
```

Finally, the module `RunTypeInference` does not currently call the F-to-Haskell translator to generate Haskell code for the result program, but if you changed it to print out the Haskell code rather than test results, you would type:

```
$ ./all Test/Working/appendF.hs -fhi-version=408 >appendFresults.hs
```

Of course, you could substitute in any Haskell program instead of `Test/Working/appendF.hs`, and you could call the output file anything else instead of `appendFresults.hs`.

There is a shell script called `hstest` in the `Test` subdirectory, which expects that `RunTypeInference` will print out the Haskell code resulting from performing deforestation on each test case (and will not print out anything else). You can run it by typing:

```
$ Test/hstest
```

For each file `foo.hs` in the `Test/Working` directory, this will create a file called `foo.hsresults.hs` in the `Test/Working/Test/Working` directory. (Changing the script to call the file something more sensible and put it in a different directory is left as an exercise for the reader.)