Extending type systems for intermodule error handling and optimization

Einar Karttunen

April 19, 2004

Abstract

Functional programming is a programming paradigm that emphasises the evaluation of expressions. Many functional languages are pure, which means that expressions behave like mathematical functions. If the expressions are pure the order of evaluation is not significant and it can be performed only when needed i.e. lazily. This paper represents alternative ways of optimising lazy, statically typed languages.

Common problems include trivial update problem and intermodule optimisations in face of separate compilation. We use a Hindley-Milner type system extended with rank-2 polymorphic types, but the ideas could be applied to much wider scale of languages.

Contents

1 Evaluating a lazy language

Lazy statically typed functional languages are usually compiled to a small core language containing few primitives. These include top-level bindings, let, case, constructors and a lambda expression form. The core language is compiled for an abstract machine. Values are either builtins e.g. integers, unevaluated thunks or evaluated values. Some representations like the Spineless Tagless G-machine[Jon92] use a common representation for both evaluated and unevaluated values.

Usually values are evaluated only if they are needed in a *case* statement. Using this approach naively can lead to space leaks. Functional languages usually add annotations for the programmer to mark parts of the programs as strict, which is needed to get good performance in many corner cases.

2 Type attributes

Marking behaviour of code inside the language has several benefits. Interface descriptions are portable across compilers and foreign functions can be annotated. Marking them only inside the compiler gives no real benefit and simpler compiler implementations can ignore them.

2.1 Termination

Normally when evaluating a purely functional program thunks are replaced with black holes at the beginning of the evaluation. This can be optimised by marking black holes from the update list only upon garbage collection[Jon92]. Black holes are marked to discover nonterminating computation, which can then be flagged as $\perp$ and the evaluation continued.

If we knew which computations would terminate blackholing them would be unnecessary. This would make also the update stack much smaller.

It is in general impossible to know whether a given function will terminate or not. We do know it however for many functions. A function will terminate if it calls only functions that are known to terminate. This naive definition fails for (mutually) recursive functions, higher order functions and so on. Even more functions could be labelled as terminating with more extensive program analysis and specialisation [1].

Also terminating values can always be evaluated strictly which in turn makes other optimisations possible.

[1] e.g. generating different versions of HOFs based on arguments

2.2 Aliasing

Even more important than flagging termination with type attributes is marking how values are aliased. The interesting question is *"How many references does the result have to the parameter or parts of it"*. This is enough because the functions are pure i.e. they have no side effects. As the language is lazy all function applications potentially create a thunk with a reference to the parameters.

A classic problem in functional languages is the trivial update problem. A large datastructure is modified in a trivial way which results in an expensive copy-operation. What if we knew when it would be safe to actually update the structure in place? Other uses include releasing memory that is not needed anymore in a precise matter. This should reduce heap usage and garbage collection overhead.

The uniqueness types in Clean are an explicit version of this. They are limited to only expressing whether a value is unique or shared. Even this opens a can of worms and creates subtly complex rules for attribute propagation.

3 Lazy Evaluation

Lazy evaluation makes many optimisations possible. For example concatenating lists becomes effectively $O(1)$. This is possible as we can defer the actual evaluation. In practise concatenation produces a "list of lists" which is then used automatically.

This makes many seemingly inefficient solutions quite efficient and makes it possible to express algorithms in simple forms, while still retaining good performance.

While laziness gives some performance with deferred operations, it also causes very many problems. In practise one is forced to mark sections of programs to be evaluated strictly to avoid space leaks.

4 Monads

Haskell's solution to the trivial update problem are monads. A monad is a construct from category theory following a set of monad laws. They specify how different values of a monadic type may be combined. In haskell all destructive operations are performed with monads.

Monads are used for input/output, statefull computations, parsers and even lists can be thought of as monads. The monads we are interested in

wrap a computation that can be handled in an abstract matter.

4.1 Haskell ST Monad

Haskell implementations provide means of modelling state with the ST monad. This can be done transparently inside functions as [LJ94] suggests, but needs a rank-2 polymorphic type. This is needed to safely encapsulate the state transformer. The ST monad is supported in both most popular haskell implementations: GHC and Hugs.

Although the code using ST monad is clean from outside, the code inside needs to be imperative and is quite tedious to write. Also interleaving monads is quite hard. It can be done with monad combinators, but is nontrivial and leads to quite complicated code [2].

Future work may make code using monad transformers a very viable alternative but currently they lack maturity and ease of use.

4.2 Haskell IO Monad

All haskell code interfacing with the outside world, be it FFI with state, real IO, or concurrency uses IO. Most of the mutable data structures in the haskell standard library use IO to keep reference of state. All modern haskell implementations use IO as the whole input/output system is based on it.

IO can be thought as a special case of ST, but unlike ST it cannot be contained inside a function. All code using IO will have a tainted type signature. [3]

IO is used instead of ST mainly because mixing the monads is thought to be complex. This has the unfortunate effect that the monadic signatures spread further then they would need.

Using a simple dictionary from the IO monad is quite simple. All the operations return a monadic value and are thus "tainted".

```
data HashTable key val
new :: (key -> key -> Bool) -> (key -> Int32) -> IO (HashTable key val)
insert :: HashTable key val -> key -> val -> IO ()
delete :: HashTable key val -> key -> IO ()
lookup :: HashTable key val -> key -> IO (Maybe val)
..
```

[2] Some may argue that the code is elegant, but it is very hard to understand in practise

[3] Haskell does provide unsafe functions for this like `unsafePerformIO: IO a → a`, both those are generally frowned upon.

```
stuff ht key = do val <- Data.HashTable.lookup ht key
                  case val of
                       Nothing  -> return 0
                       (Just a) -> return a
```

The example just looks up a key in dictionary and returns the value if found, 0 otherwise. This is safe as one generally cannot get rid of the IO type and it forces correct sequencing of all the operations.

5 Uniqueness typing

Clean uses type attributes to properly interleave statefull and functional computations. A value with an *unique* type must be used always once.

Uniqueness typing makes it easy to make state local inside a given function without any trace of it outside it. Combining unique values of different kinds is easy and the system seems easy and intuitive at the first glance. Uniqueness typing solves the trivial update problem when the datastructure is not shared.

Uniqueness grows usually to outside. This is needed to keep the system safe as aliasing a structure containing the unique value would break the uniqueness property.

However the type attributes make the system more complex forcing complicated rules on resolving typing with higher order functions and polymorphism. Also the use of uniqueness typing for e.g. files may not be the best approach. The mathematical models for uniqueness typing seem also more complex than monads, which have a very sound mathematical foundation.

```
ScaleArrayElem:: *{#Real} Int Real -> .{#Real}
ScaleArrayElem ar i factor
# (elem,ar) = ar![i]
= {ar & [i] = elem*factor}
```

This Clean example takes an unique array, an index and a scale factor as parameters and scales one array element. This can be implemented using destructive updates as the input is unique. Even in this simple case the type signature is nontrivial. Automatically inferred uniqueness typing information would seem like a very powerful tool.

6 Automatic analysis

A more transparent system than either monads or uniqueness typing would be necessary to make the optimisations easy for novice programmers. Monadic code, while useful for other purposes is clearly too convoluted for solving the update problem.

Various automatic optimisation schemes have been suggested. They usually rely on rewrite sequences using few primitive operations which are used to define library functions. These optimisations usually rely on specific code transformations and are thus quite brittle to changes in the code. A small insignificant change may alter the performance of an algorithm significantly based on optimiser decisions.

Recursive definitions are often quite hard of the optimisations. This is a large problem as functional languages use recursion for very many tasks. Also the efficiency of library functions is very sensitive for their definitions.

For example to calculate the sum of the elements one could define the following functions:

```
-- builtin way
sum0 :: [Int] -> Int
sum0 xs = sum xs

-- fold with builtin
sum1 :: [Int] -> Int
sum1 xs = foldr (+) 0 xs

-- fold with lambda
sum2 :: [Int] -> Int
sum2 xs = foldr (\a b -> a + b) 0 xs

-- recursive
sum3 :: [Int] -> Int
sum3 []     = 0
sum3 (x:xs) = x + sum3 xs

-- foldr1
sum4 :: [Int] -> Int
sum4 xs = foldr1 xs
```

All the definitions *should* produce code that performs adequately. In reality their performance might vary drastically from implementation to im-

plementation. This is clearly unacceptable. With e.g. GHC 6.2.2 *sum1* is the fastest with a quite wide margin. Some of the implementations are even ten times slower than *sum1*. If the types are left for the compiler to infer it uses a wider type (Num) and actually produces faster code for sum2, which was ten times slower than sum1 with the added type signature.

7 Deforestation

Deforestation means removing intermediate structures from a program without changing semantics. This is particulary easy in non-strict functional languages. Calculating the sum of the cubes of the numbers from 1 to n could be expressed in haskell as:

```
cube_sum n = sum [ i*i | i <- [1..n]]
```

The code constructs a list containing the numbers from 1 to n, and from that produces a list containing the cube of each element. Finally the elements are summed together. The goal is to create a program transformation which removes the intermediate lists.

In practise the techniques employed today are quite rudimentary and in many cases dependent on using builtin list folding (*foldr*[4]) and build functions. As all library functions use them this is quite painless, but making the transformations for recursive definitions is hard. *Let* and *case* constructs need to be pushed outside the fold/build, but this is a worhwile optimasation in itself.

A simple rewriting system is introduced in [GJ94]. It is based on two primitive operations: foldr and build. Build is a simple list constructor with a naive definition and inferred type:

```
build :: ((a -> [a] -> [a]) -> [b] -> c) -> c
build x = x (:) []
```

To remove the structures we define a rewriting rule:

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

This is clearly works only if g uses it's arguments, this can be forced with a rank-2 typing [5]. This requirement is essentially the same as the one required for the ST monad.

[4] foldr:: (a -> b -> b) -> b -> [a] -> b

[5] This gives the typing system more power, without making inference harder than with Hindley-Milner. In this case we need the non-toplevel universal quantifier.

```
build :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
build x = x (:) []
```

This allows the transformation to be performed safely iff the left hand side of the rewriting rule is well typed. Deforestation makes it more easy to provide clean functional interfaces without sacrificing performance. The optimisations can be applied even in intermodule contexts, but special care must be taken to interleave inlining and deforestation in the right order.

Deforestation has been implemented in the *Glasgow Haskell Compiler* and detailed analysis can be found in [Gil96]. Test results show that simple deforestation With cheap deforestation compile times rose 4% on average. The resulting binaries were 6% smaller than the original ones. The optimisation dropped the instruction counts and heap allocation and residency by 5%, while growing maximum stack usage.

References

[Gil96] A. Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Glasgow University, Glasgow, Scotland, UK, 1996.

[GJ94] Andrew J. Gill and Simon L. Peyton Jones. Cheap deforestation in practice: An optimiser for haskell. 1994.

[Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[LJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994.