Replace this file with `prentcsmacro.sty` for your meeting, or with `entcsmacro.sty` for your meeting. Both can be found at the ENTCS Macro Home Page.

Using Template Haskell for Abstract Interpretation

Clara Segura[1],[2]

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Madrid, Spain

Carmen Torrano[3]

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Madrid, Spain

Abstract

Metaprogramming consists of writing programs that generate or manipulate other programs. Template Haskell is a recent extension of Haskell, currently implemented in the Glasgow Haskell Compiler, giving support to metaprogramming at compile time. Our aim is to apply these facilities in order to statically analyse programs and transform them at compile time. In this paper we use Template Haskell to implement an abstract interpretation based strictness analysis and a let-to-case transformation that uses the results of the analysis. This work shows the usefulness of the tool in order to incorporate new analyses and transformations into the compiler without modifying it.

Keywords: Meta-programming, abstract interpretation, strictness analysis.

1 Introduction

Metaprogramming consists of writing programs that generate or manipulate other programs. Template Haskell [17,18] is a recent extension of Haskell, currently implemented in the Glasgow Haskell Compiler [12] (GHC), giving support to metaprogramming at compile time. Its functionality is obtained from the library package `Language.Haskell.TH`. It has been shown to be a useful tool for different purposes [6], like program transformations [7] or the definition of an interface for Haskell with external libraries (http://www.haskell.org/greencard/). Specially interesting is the implementation of a compiler for the parallel functional language Eden [15] without modifying GHC.

[1] Work partially supported by the Spanish project TIN2004-07943-C04.

[2] Email: `csegura@sip.ucm.es`

[3] Email: `ctorranog@hotmail.com`

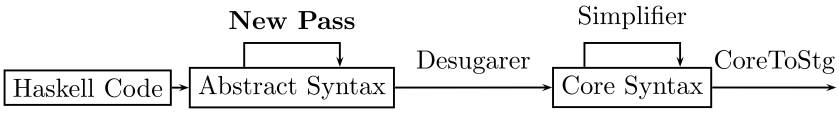

Fig. 1. GHC compilation process with new analyses and transformations

Using such extension, a program written by a programmer can be inspected and/or modified at compile time before proceeding with the rest of the compilation process. Our aim is to apply these metaprogramming facilities in order to statically analyse programs and transform them at compile time. This will allow us on the one hand to quickly implement new analyses defined for functional languages and on the other hand to incorporate these analyses into the compiler without modifying it. In Figure 1 we show an scheme of GHC compilation process. Haskell code is desugared into a simpler functional language called Core. Analyses and transformations in GHC take place at Core syntax level, which are summarized as a *simplifier* phase. In order to add new analyses and transformations it would be necessary to modify the compiler. However, by using Template Haskell these can be incorporated at the level of Haskell syntax without modifying GHC. In Figure 1 this is added as a new pass at the level of the abstract syntax tree.

In particular, languages like Eden [5] can benefit from these facilities. Eden is a parallel extension of Haskell whose compiler is implemented on GHC [3]. Several analyses have been theoretically defined for this language [14,11,4] but they have not been incorporated to the compiler because this involves the modification of GHC, once for each new analysis we could define, which seems unreasonable. Using Template Haskell new analyses and/or transformations could be first prototyped and then incorporated to the compilation process without directly modifying the internals of the compiler.

In this paper we explore the usefulness of Template Haskell for these purposes by implementing an abstract interpretation based strictness analysis and a let-to-case transformation that uses the results of the analysis. These are well-known and already solved problems, which allows us to concentrate on the problems arising from the tool. In Section 2 we describe those features of Template Haskell used in later sections. In Section 3 we give an introduction to abstract interpretation, and describe the strictness analysis and the let-to-case transformation. Section 4 describes their implementation using Template Haskell and shows some examples. Finally, in Section 5 we conclude and discuss the improvements to the tool that could make it more useful.

2 Template Haskell

Template Haskell is a recent extension of Haskell for compile-time meta-programming. This extension allows the programmer to observe the structure of the code of a program and either transform that code, generate new code from it, or analyse its properties. In this section we summarize the facilities offered by the extension.

The code of a Haskell expression is represented by an algebraic data type `Exp`, and similarly are represented each of the syntatic categories of a Haskell program, like declarations (`Dec`) or patterns (`Pat`). In Figure 2 we show parts of the definitions of these data types, which we will use later in Section 4.

```
data Exp =
    LitE Lit                  -- literal
    VarE Name                 -- variable
    ConE Name                 -- constructor
    LamE [Pat] Exp            -- lambda abstraction
    AppE Exp Exp              -- application
    CondE Exp Exp Exp         -- conditional
    LetE [Dec] Exp            -- let expression
    CaseE Exp [Match]         -- case expression
    InfixE (Maybe Exp) Exp (Maybe Exp) -- primitive op.
    . . .
data Match =
    Match Pat Body [Dec]   -- pat -> body where decs
data Pat =
    VarP Name            -- variable
    ConP Name [Pat]      -- constructor
    . . .
data Body =
    NormalB Exp -- just an expression
    . . .
data Dec =
   ValD Pat Body [Dec]                     -- v = e where decs
   FunD Name [Clause [Pat] Body [Dec]]     -- f p1 ... pn = e
                                           --    where decs
```

Fig. 2. Data types representing Haskell syntax

A quasi-quotation mechanism allows one to represent templates, i.e. Haskell programs at compile time. Quasi-quotations are constructed by placing brackets, `[|` and `|]`, around concrete Haskell syntax fragments, e.g. `[|\x->x|]`.

This mechanism is built on top of a monadic library. The quotation monad `Q` encapsulates meta-programming features as fresh name generation:

```
instance Monad Q
runQ :: Q a -> IO a   . . .
```

The usual monadic operators `bind`, `return` and `fail` are available, as well as the do-notation [19]. This is everything we need to know about the quotation monad for our purposes.

The translation of quoted Haskell code makes available it abstract syntax tree as a value of type `ExpQ`, where `type ExpQ = Q Exp`; e.g. `[|\x->x|]::ExpQ`.

Library `Language.Haskell.TH` makes available syntax construction functions built on top of the quotation monad. Their names are the same as the constructors of the algebraic data types, but lower case, e.g. `lamE :: [PatQ] -> ExpQ -> ExpQ`. For example, we can build expression `[|\x->x|]` also by writing `lamE [varP (mkName "x")] (varE (mkName "x"))`, where `mkName:: String -> Name`.

Evaluation can happen at compile time by means of the splice notation `$`. This means compile-time evaluation when placed at top level. Also, when placed inside a quasi-quoted expression it means evaluation when the quasi quoted code is constructed. The result of such evaluation is spliced into the enclosing expression. As an example, `[|\x->$e|]` evaluates `e` at compile time and the result of the evaluation, a Haskell expression `e'`, is spliced into the lambda abstraction giving `[|\x->e'|]`.

We will use in Section 4 the quasi-quotation mechanism in order to analyse and transform Haskell programs, and the splicing notation in order to do this at compile time. A pretty printing library `Language.Haskell.TH.PprLib` will be useful in order to visualize the results of our examples.

There are other features of Template Haskell we are not using here; the interested reader may look at [17,18] for more details.

3 Strictness Analysis and let-to-case transformation

3.1 Motivation

Practical implementations of functional languages like Haskell use a call-by-need parameter passing mechanism. A parameter is evaluated only if it is used in the body of the function; once it has been evaluated to weak-head normal form, it is updated with the new value so that subsequent accesses to that parameter do not evaluate it from scratch. The implementation of this mechanism builds a closure or suspension for the actual argument, which is updated when evaluated. The same happens with a variable bound by a let expression: A closure is built and it is evaluated and subsequently updated when the main expression demands its value.

Strictness analysis [9,1,20,2] detects parameters that will be evaluated by the body of a function. In that case the closure construction can be avoided and its evaluation can be done immediately. This means that call-by-need is replaced by call-by-value.

The same analysis can be used to detect those variables bound by a let expression that will be evaluated by the main expression of the let. Such variables can be immediately evaluated, so that the let expression can be transformed into a case expression without modifying the expression semantics [16]. This is known as *let-to-case* transformation:

$$\textbf{let } x = e \textbf{ in } e' \Rightarrow \textbf{case } e \textbf{ of } x \rightarrow e'$$

Notice that this transformation assumes a strict semantics for the case expression. Core case expression is strict in the discriminant, but Haskell case with a unique variable pattern alternative is lazy. As our analysis and transformation happen at Haskell level we would not obtain the desired effect with the previous transformation. Additionally it can even be incorrect from the point of view of the types because let-bound variables are polymorphic while case-bound ones are monomorphic.For example, the expression $\textbf{let } x = [\,] \textbf{ in case } x \textbf{ of } [\,] \rightarrow (1 : x, 'a' : x)$ is type correct while its transformed version is not.

However we can use Haskell's polymorphic infix `seq::a->b->b` operator to obtain the desired effect maintaining the types. It evaluates its first argument to weak head normal form and then returns as result its second argument. Consequently, our transformation is the following: $\textbf{let } x = e \textbf{ in } e' \Rightarrow \textbf{let } x = e \textbf{ in } x \textit{ seq } e'$

3.2 Strictness Analysis by Abstract Interpretation

Strictness analysis can be done by using abstract interpretation [10]. This technique can be considered as a non-standard semantics in which the domain of values is replaced by a domain of values descriptions, and where each syntactic operator is given a non-standard interpretation allowing to approximate at compile time the run-time behavior with respect to the property being studied.

Mycroft [9] gave for the first time an abstract interpretation based strictness

$$
\begin{array}{rlll}
e \rightarrow & c & & \{\text{ constant }\} \\
& \mid v & & \{\text{ variable }\} \\
& \mid e_1 \ op \ e_2 & & \{\text{ primitive operator }\} \\
& \mid \mathit{if} \ e_1 \ \mathit{then} \ e_2 \ \mathit{then} \ e_3 & & \{\text{ conditional}\} \\
& \mid \lambda b.e & & \{\text{ first-order lambda }\} \\
& \mid C \ e_1 \ldots e_n & & \{\text{constructor application }\} \\
& \mid e_1 \ e_2 & & \{\text{ function application }\} \\
& \mid \mathbf{let} \ v_1 = e_1 \ldots v_n = e_n \ \mathbf{in} \ e & & \{\text{ let expression }\} \\
& \mid \mathbf{case} \ e \ \mathbf{of} \ alt_1 \ldots alt_n & & \{\text{ case expression }\} \\
alt \rightarrow & C \ b_1 \ldots b_n \rightarrow e & & \\
& \mid b \rightarrow e & &
\end{array}
$$

Fig. 3. A first-order subset of Haskell

analysis for a first-order functional language. Later, Burn et al. [1] extended it to higher order programs and Wadler [20] introduced the analysis of data types. Peyton Jones and Partain [13] described how to use signatures in order to make abstract interpretation more efficient.

We show here an abstract interpretation based strictness analysis for expressions of a first-order subset of Haskell with data types, whose syntax is shown in Figure 3. For the moment, this analysis is enough for our purposes. In Section 5 we discuss the extension of the analysis to higher order and in general to full Haskell.

Notice that for flexibility reasons we allow lambda abstractions as expressions, but we restrict them to be first-order lambda abstractions, i.e. the parameter is a variable b that can only be bound to a zeroth order expression.

As the language is first-order the only places where lambda abstractions are allowed are function applications and right hand sides of let bindings. Function and constructor applications must be saturated. Let bindings may be recursive. Notice that if we lift the previously mentioned restrictions we have a higher-order subset of Haskell. This is the reason for our definition.

Case expressions may have at most one default alternative ($b \rightarrow e$).

The basic abstract values are $\perp$ and $\top$, respectively representing strictness and "don't know" values, where $\perp \leq \top$. Operators $\sqcap$ and $\sqcup$ are respectively the greatest lower bound and the least upper bound. In order to represent the strictness of a function in its different arguments we use abstract functions over basic abstract values a. For example $\lambda a_1.\lambda a_2.a_1 \sqcap a_2$ represents that the function is strict in both arguments, and $\lambda a_1.\lambda a_2.a_1$ represents that it is strict in its first argument but that we do not know anything about the second one.

In Figure 4 we show the interpretation of each of the language expressions, where ρ represents an abstract environment assigning abstract values to variables. The environment $\rho + [v \rightarrow av]$ either extends environment ρ if variable v had no assigned abstract value, or updates the abstract value of v if it had. The interpretation is standard so we only give some details.

$$
\begin{aligned}
&[\![c]\!]\ \rho = \top \\
&[\![v]\!]\ \rho = \rho(v) \\
&[\![e_1\ op\ e_2]\!]\ \rho = [\![e_1]\!]\ \rho \sqcap [\![e_2]\!]\ \rho \\
&[\![\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{then}\ e_3]\!]\ \rho = [\![e_1]\!]\ \rho \sqcap ([\![e_2]\!]\ \rho \sqcup [\![e_3]\!]\ \rho) \\
&[\![\lambda b.e]\!]\ \rho = \lambda a.[\![e]\!]\ (\rho + [b \to a]) \\
&[\![C\ e_1 \dots e_n]\!]\ \rho = \top \\
&[\![e_1\ e_2]\!]\ \rho = [\![e_1]\!]\ \rho\ [\![e_2]\!]\ \rho \\
&[\![\mathbf{let}\ v_1 = e_1 \dots v_n = e_n\ \mathbf{in}\ e]\!]\ \rho = [\![e]\!]\ \rho' \\
&\qquad\qquad \mathit{where}\ \rho' = \mathit{fix}\ f \\
&\qquad\qquad\qquad f = \lambda\rho.\rho + [v_1 \to [\![e_1]\!]\ \rho, \dots v_n \to [\![e_n]\!]\ \rho] \\
&[\![\mathbf{case}\ e\ \mathbf{of}\ b \to e']\!]\ \rho = [\![e']\!]\ (\rho + [b \to a]) \\
&\qquad\qquad \mathit{where}\ a = [\![e]\!]\ \rho \\
&[\![\mathbf{case}\ e\ \mathbf{of}\ \mathit{alt}_1 \dots \mathit{alt}_n]\!]\ \rho = a \sqcap (a_1 \sqcup \dots \sqcup a_n) \qquad (n > 1) \\
&\qquad\qquad \mathit{where}\ a = [\![e]\!]\ \rho \\
&\qquad\qquad\qquad a_i = [\![\mathit{alt}_i]\!]\ \rho\ a \\
&[\![C\ b_1 \dots b_n \to e]\!]\ \rho\ a = [\![e]\!]\ (\rho + [b_1 \to a, \dots b_n \to a]) \\
&[\![b \to e]\!]\ \rho\ a = [\![e]\!]\ (\rho + [b \to a])
\end{aligned}
$$

Fig. 4. A strictness analysis by abstract interpretation

Primitive binary operators, like $+$ or $*$, are strict in both arguments so we use $\sqcap$ operator. The abstract value of a constructor application is $\top$ because constructors are lazy. This means for example, that function $\lambda x.x : [\,]$ is not considered strict in its first argument. Notice that in the lists abstract domain we have safely collapsed the four-valued abstract domain of Wadler [20] into a two-valued domain, where for example $\bot : \bot$, $[1, \bot, 2]$ and $[1, 2, 3]$ are abstracted to $\top$, and only $\bot$ is abstracted to $\bot$. In the three examples it is safe to evaluate the list to weak head normal form.

In a case expression the variables bound by the case alternatives inherit the abstract value of the discriminant. When there is only a default alternative the case is lazy, otherwise is strict in the discriminant.

As we have used first-order abstract functions as abstract values, function application can be easily interpreted as abstract function application. To interpret a let expression we need a standard fixpoint calculation as it may be recursive.

3.3 Signatures

Abstract interpretation based analyses of higher order functions is expensive. Signatures [13] can be used in order to improve their efficiency although they imply losing some precision in the analysis. We use them in our implementation as we are interested in analyses for full Haskell. Strictness basic signatures are $\bot$ and $\top$. Signatures for functions of n arguments are n-tuples of signatures $(s_1, \dots, s_n)$

indicating whether the function is strict in each of its arguments. For example, $(\perp, \top, \perp)$ is the signature of a function with three arguments that is strict is the first and the third arguments.

The strictness signature of a function is obtained by probing it with n combinations of arguments. Component s_i is calculated by applying the function to the combination in which the ith argument is given the value $\perp$ and the rest of them are given the value $\top$. For example, the signature of function $\lambda x.\lambda y.\lambda z.x+y$, $(\perp, \perp, \top)$, is obtained by applying the function to $(\perp, \top, \top)$, $(\top, \perp, \top)$ and $(\top, \top, \perp)$.

When considering higher order, functions must be probed with signatures of the appropriate functional types. For example in $\lambda f.\lambda x.f\ 3+x$, the first argument is a function, so it has to be probed with $((\perp, \perp), \top)$ and $((\top, \top), \perp)$ giving $(\perp, \perp)$, as expected. In Section 5 we will discuss about the problems encountered in this case, when trying to extend the analysis.

4 Implementation using Template Haskell

In this section we describe the implementation of the strictness analysis and the corresponding transformation using Template Haskell. Given a Haskell expression `e` the programmer wants to evaluate, this is the module he/she has to write:

```
module Main where
import Strict
import System.IO
import Language.Haskell.TH

main = putStr (show $(transfm  [| e |]))
```

Module `Strict` contains the transformation function and the strictness analysis. First we quote the Haskell expression in order to be able to inspect the abstract syntax tree; then we modify such tree using function `transfm`, defined below. We use `$` to execute the transformation at compile time.

These small modifications could be even completely transparent to the programmer if we generate them automatically. If we want the new pass to do more things we just have to modify function `transfm`.

4.1 Strictness Analysis Implementation

The analysis is carried out by function `strict :: Exp -> Env -> AbsVal` which given an expression and a strictness environment returns the abstract value of the expression. Abstract values are represented using a data type `AbsVal`:

```
data StrictAnnot = Bot  |  Top deriving (Show,Eq)
data AbsVal = B StrictAnnot  |  F [StrictAnnot]   | FB Int
```

The basic annotations are `B Bot`, to represent strictness, and `B Top` to represent the "don't know" value. The abstract value of a function with n arguments is approximated through a signature of the form `F [b1, b2, ..., bn]` where each `bi` indicates whether the function is strict in the ith argument. The special `FB n` value is the abstract value of a completely undefined function with n arguments,

```
strict :: Exp -> Env -> AbsVal
strict (VarE s) rho = getEnv s rho
strict (LitE l) rho = B Top
strict (InfixE (Just e1) e (Just e2)) rho =
  if (isCon e) then (B Top)
  else inf (strict e1 rho) (strict e2 rho)
strict (CondE e1 e2 e3) rho =
  inf (strict e1 rho)
      (sup (strict e2 rho) (strict e3 rho))
```

Fig. 5. Strictness Analysis Implementation-Basic Cases

```
strict (LamE ((VarP s):[]) e) rho =
  let B b = strictaux e (addEnv (s,0,B Bot) rho) in
  case (strict e (addEnv (s,B Top) rho)) of
    B b1 -> F (b:[])
    F bs -> F (b:bs)

strictaux::Exp -> Env -> AbsVal
strictaux (LamE ((VarP s):[]) e) rho =
    strictaux e (addEnv (s,B Top) rho)
strictaux e rho = strict e rho
```

Fig. 6. Strictness Analysis Implementation-Lambda Expressions

that is, the bottom of the functional abstract domain, which is useful in several places.

The transformation function calls this function, but if we want to prove the prototype with examples we can write the following:

```
main = putStr (show $(strict2 [| e |] empty))
```

where `e` is a closed expression we want to analyse, `empty` represents the empty strictness environment, and function `strict2` is defined as follows:

```
strict2 :: ExpQ -> Env -> ExpQ
strict2 eq rho = do {e <- eq ;
                     return (toExp(strict e rho))}
```

where function `toExp :: AbsVal -> Exp` just converts an abstract value into an expression. Notice that the analysis is carried out at compile time and that we have defined `strict2` as a transformation from a expression to another expression representing its abstract value. This is because the compile time computations happen inside the quotation monad, so both the argument and the result of `strict2` must be of type `ExpQ`. We use do-notation in order to encapsulate `strict` into the monadic world.

Function `strict` is the actual strictness analysis defined by cases over the syntax, we need to remember the `Exp` data type definition (shown in Figure 2) and the restrictions of our language (explained in the previous section).

In Figure 5 we show the interpretation of constants, primitive operators, variables and conditional expressions, as shown in the previous section. We have to be careful with infix operators because some constructors like lists `:` are infix. We distinguish them using function `isCon`, which we do not show here. Operator `inf` calculates the greatest lower bound and `sup` the least upper bound, and `getEnv` gets from the environment the abstract value of a variable.

In Figure 6 we show the interpretation of a lambda abstraction. Its value is a signature `F [b1, ..., bn]`, being `n` the number of arguments, obtained by probing the function with several combination of arguments, as we explained in Section 3.3. We start probing the function with the first argument. First, we give it the value

```
strict (ConE cons) rho = B Top
strict (AppE (ConE cons) e) rho = B Top
strict (AppE e1 e2) rho =
  if (isCon e1) then B Top
  else absapply (strict e1 rho) (strict e2 rho)

absapply::AbsVal -> AbsVal -> AbsVal
absapply (FB n) a
         | n==1 = B Bot
         | n > 1 = FB (n-1)

absapply (F (h:tl)) (B b)
         | null tl = B x
         | x == Top = F tl
         | otherwise = FB (length tl)
         where x = sups h b
```

Fig. 7. Strictness Analysis Implementation-Applications

`B Bot` and the auxiliary function `strictaux` gives the rest of the arguments the value `B Top`. Then we give it the value `B Top` and recursively probe with the rest of the arguments. In such a way we obtain all the combinations we wish.

In Figure 7 we show the interpretation of both constructor and function applications. From the point of view of the language they are the same kind of expression, so we use again function `isCon` to distinguish them.

If it is a function application, `absapply` carries out the abstract function application. The abstract value `FB n` represents the completely undefined function so it returns `B Bot` when completely applied and `FB (n-1)` when there are remaining arguments to be applied to.

When a signature `F [b1, ..., bn]` is applied to an abstract value `B b` we need to know whether it is the last argument. If that is the case we can return a basic value, otherwise we have to return a functional value. The resulting abstract value depends on both `b1` and `b`.

If `b1` is `Top` the function is not necessarily strict in its first argument, so independently of the value of `b` we can return `B Top` if it was the last argument or continue applying the function to the rest of the arguments by returning the rest of the list.

The same happens if `b` is `Top` as `head xs` was obtained by giving the first argument the value `Bot`: we have lost information and the only thing we can say is "we don't know" and consequently either return `B Top` or continue applying the function.

If neither `b1` nor `b` is `Top` (i.e. when the least upper bound `sups` returns `Bot`) then the function is strict in its first argument, which is undefined, so we can return `B Bot` independently of the rest of the arguments. However if there are arguments left we return the completely undefined function `FB (n-1)`.

In Figure 8 we show the interpretation of a let expression. Auxiliary function `strictdecs` carries out the fixpoint calculation. Function `splitDecs` splits the left hand sides (i.e. the bound variables) and the right hand sides of the declarations. The initial environment `init` is built by extending the environment with the new variables bound to an undefined abstract value of the appropriate type, done by `extendEnv`. Function `combines` updates the environment with the new abstract values in each fixpoint step; it also returns a boolean value `False` when the environment does not change and consequently the fixpoint has been reached.

Finally, in Figure 9 we show the interpretation of a case expression. Function `nostrict` returns true if it is a lazy case expression. The first two branches of `casealt` correspond to constructor pattern matches (either infix or prefix) and the

```
strict (LetE ds e) rho = strict e (strictdecs ds rho)

strictdecs:: [Dec] -> Env -> Env
strictdecs [ ] rho = rho
strictdecs ds rho =
 let
    (varns,es) = splitDecs ds
    init = extendEnv rho varns
    f = \ rho' ->let
                    aes = map (flip strict rho') es
                    triples = zipWith triple varns aes
                  in
                    combines rho' triples
    fix g (env,True) = fix g (g env)
    fix g (env,False) = env
 in
   fix f (init,True)
```

Fig. 8. Strictness Analysis Implementation-Let Expressions

```
strict (CaseE e ms) rho =
                     let
                       se = strict e rho
                        l = caseaux ms se rho
                       sl = suplist l
                     in
                       if (nostrict ms) then sl
                       else (inf se sl)


caseaux :: [Match] -> AbsVal -> Env -> [AbsVal]
caseaux ms se rho = map (casealt se rho) ms

casealt :: AbsVal -> Env -> Match -> AbsVal
casealt abs rho m =
  case m of
    Match (InfixP (VarP h) con (VarP tl)) (NormalB e) [] ->
         let rho' = addEnvPat abs [VarP h, VarP tl] rho
         in strict e rho'
    Match (ConP con ps) (NormalB e) []->
         let rho' = addEnvPat abs ps rho
         in  strict e rho'
    Match (VarP x)(NormalB e)[] ->
         let rho' = addEnvPat abs ((VarP x):[]) rho
         in strict e rho'
```

Fig. 9. Strictness Analysis Implementation-Case Expressions

third one to the variable alternative. Function `suplist` calculates the least upper bound of the alternatives, and `casealt` interprets each of the alternatives. The variables bound by the case alternatives inherit the abstract value of the discriminant, which is done by function `addEnvPat`.

Example 4.1 Given the expression `\ x -> \ y -> 3 * x` , the analysis returns `F [Bot, Top]`, as expected; i.e. the function is strict in the first argument.

Example 4.2 Another example with a case expression is the following one:

```
\ x -> \ z-> case 1:[] of [] -> x
                          y:ys -> x + z
```

The result is `F [Bot, Top]` as expected, telling us that the function is strict in the first argument but maybe not in the second one, although we know it is. Notice the loss of precision. This is because the analysis is static, but not because of the implementation.

Example 4.3 The use of signatures in the implementation implies a loss of precision with respect to the analysis shown in Section 3. For example, function

```
\ x -> \ y -> \ z -> if z then x else y
```

has abstract value $\lambda a_1.\lambda a_2.\lambda a_3.a_3 \sqcap (a_1 \sqcup a_2)$ but the implementation would assign

```
transf :: Exp -> Env -> Exp
transf (LetE ds e) rho =
 if (isRecorFun ds) then
   let
      (vs,es) = splitDecs ds
      rho' =  foldr addEnvtop rho vs
      es' = map (flip transf rho') es
      ds' = zipWith makeDec ds es'
      te' = transf e rho'
   in LetE ds' te'
 else
   case (head ds) of
      ValD (VarP x) (NormalB e') [] ->
       let
        te' = transf e' rho
        te = transf e (addEnv (x,B Top) rho)
        ds' = ValD (VarP x) (NormalB te') []:[]
        lambda = LamE ((VarP x):[]) te
        F bs = strict lambda rho
       in if (head bs) == Bot then
          LetE ds' (InfixE (Just (VarE x))
                           (VarE (mkName "Prelude:seq"))
                           (Just te))
          else LetE ds' te
```

Fig. 10. Transformation of a let expression

it signature `F [Top, Top, Bot]` which is undistinguishable from abstract value $\lambda a_1.\lambda a_2.\lambda a_3.a_3$. Function `\ x -> \ y -> \ z -> z` has the same signature.

4.2 Transformation implementation

The let-to-case transformation has been developed in a similar way. We want the transformation function to be applied not only to the main expression at top level but also, when possible, to all its subexpressions. For example, function `\ x -> let z = 3 in x + z` can be transformed to `\ x -> let z = 3 in z seq (x + z)`. But then, even when the main expression is closed, subexpressions may have free variables. Consequently, we need a strictness environment, initially empty, carrying the abstract values of the free variables:

```
transfm e = transf2 e empty
transf2 :: ExpQ -> Env -> ExpQ
transf2 eq rho = do {e <- eq;
                     return (transf e rho)}
```

In this case, if we want to view the result of the transformation instead of the evaluation of the transformed expression, we can use the function `runQ` of the monad, which allows us to extract the transformed expression before proceeding with the rest of the compilation. Then we print it with function `ppr` from the library `Language.Haskell.TH.PprLib`:

```
main = do {e <- runQ (transf2 q empty) ;
           putStr (show (ppr e))}
```

The function doing all the important work is `transf`. We show in Figure 10 only the most interesting case, the let expression. We are assuming that several definitions appearing in a let expression are mutually recursive. The compiler partitions these definitions into strongly connected components in order to benefit of polymorphism as much as possible. The content of all quasi-quoted code is typechecked [8] so it seems a reasonable assumption.

So when the let expression defines a function or is a set of recursive definitions (told by function `isRecorFun`) we do not apply the transformation at top level but we could apply it in the right hand sides of the declarations and in the main expression of the let. When transforming these expressions, the abstract values of the bound variables are irrelevant so we give them the top abstract value. This is done by `addEnvtop`.

When there is only a non-recursive binding **let** $x = e$ **in** e' we build a lambda abstraction $\lambda x.e'$ and analyse it in order to see if the body of the let is strict in the bound variable. If that is the case, the transformation is done. At the same time the right hand side of the binding and the body may also be transformed.

Example 4.4 The following expression

```
 let a = 1 in let b = 2 in a + b
```

is transformed to:

```
let a_0 = 1
 in a_0 Prelude:seq (let b_1 = 2
                      in b_1 Prelude:seq (a_0 GHC.Num.+ b_1))
```

Example 4.5 In the following example it is possible to see that the transformation may happen not only at the top level but also in any subexpression of the main expression. Function

```
\ x -> (let a = 1 in a + 3) * (let y = 2 in y + x )
```

is transformed to:

```
\ x_0 -> (let a_1 = 1 in a_1 Prelude:seq (a_1 GHC.Num.+ 3))
         GHC.Num.*
         (let y_2 = 2 in y_2 Prelude:seq (y_2 GHC.Num.+ x_0))
```

5 Conclusions and Future Work

Template Haskell is a recent extension of Haskell for metaprogramming, currently implemented in GHC 6.4.1. The design of the extension and the facilities it offers are described in detail in [17,18]. Its functionality is obtained from the library `Language.Haskell.TH`. Template Haskell has been shown to be a useful tool for different purposes [6], like program transformations [7] or the definition of an interface for Haskell with external libraries (http://www.haskell.org/greencard/). Specially interesting is the implementation of a compiler for the parallel functional language Eden [15] without modifying GHC.

In this paper we have studied how to use Template Haskell in order to incorporate new analyses and transformations to the compiler without modifying it. We have presented the implementation of a strictness analysis and a subsequent let-to-case transformation. The source code can be found at http://dalila.sip.ucm.es/miembros/clara/publications.html. These are well-known problems, which has allowed us to concentrate on the difficulties and limitations of using Template Haskell for our purposes, see the discussion below. As far as we know, this is the first time that Template Haskell has been used for developing static analyses. There are some compiling tools available for GHC (see http://www.haskell.org/libraries/#compilation) which are useful to write analyses prototypes, but our aim is to use the results of the analyses and to continue with the GHC's compilation process.

The analysis has been developed for a first-order subset of Haskell. This has been relatively easy to define. The only difficulty here is the absence of a properly commented documentation of the library. The analysis could be extended to higher-order programs. We have not done this for the moment for the following reason. When analysing higher order functions, it is necessary to probe functions with functional signatures, which we have to generate, as we explained in Section 3.3. In order to generate such signatures we need to know how many arguments the function has, which in the first order case was trivial (we just counted the lambdas) but not in the higher order case due to partial applications. If we had types available in the abstract syntax tree, it would be trivial again. In this analysis the probing signatures are quite simple; if the argument function has n arguments then the probing signature is `FB n`. But in other analyses, like non-determinism analysis [14], probing signatures are more complex and types are fundamental to generate them properly.

Although there is a typing algorithm for Template Haskell [8], the type information is not kept in the abstract tree. We could of course develop our own typing algorithm but it would be of no help for other users if it is not integrated in the tool. This would be very useful also to do type-based analyses, which we plan to investigate.

Using Template Haskell for analyses and transformations has several disadvantages. First, the analysis and transformation must be defined for full Haskell. Analyses and transformations are usually done over a simplified language where the syntactic sugar has disappeared (Core in GHC). Of course this would make sense if it were possible to control in which phase of the compiler we want to access the abstract syntax tree, and for the moment this is not the case. If the analysis is defined for a subset of Haskell, like ours, it would be necessary to study the transformations done by GHC's desugarer in order to determine how to analyse the sugared expressions. An analysis at the very beginning of the compilation process is still useful when we want to give information to the user about the results of the analysis. In that case we want to reference the original variables written by him/her, which are usually lost in further phases of the compiler. Notice that in our examples variables are indexed but they still maintain the original string name. The desugarer however generates fresh variables unknown for the programmer.

Second, we can profit only of those analyses whose results are used by a subsequent transformation. The results of the analysis cannot be propagated to further phases of the compiler, which would be affected by them. Examples of this situation is the non-determinism analysis [14] whose results are used to deactivate some transformations done by the simplifier, or the usage analysis which affects to the STG code generated by the compiler [21].

However it is useful for developing abstract interpretation based analyses whose results can be used to transform Haskell code, and incorporate easily such transformation to the compilation process.

References

[1] G. L. Burn, C. L. Hankin, and S. Abramsky. The Theory of Strictness Analysis for Higher Order Functions. In *Programs as Data Objects*, volume 217 of *LNCS*, pages 42–62. Springer-Verlag, October 1986.

[2] T. P. Jensen. Strictness Analysis in Logical Form. In R. J. M. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 352–366. Springer-Verlag, New York, 1991.

[3] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Selected Papers of the 10th International Workshop on Implementation of Functional Languages, IFL'98*, volume 1595 of *LNCS*, pages 103–119. Springer-Verlag, 1999.

[4] U. Klusik, R. Peña, and C. Segura. Bypassing of Channels in Eden. In P. Trinder, G. Michaelson, and H.-W. Loidl, editors, *Trends in Functional Programming. Selected Papers of the 1st Scottish Functional Programming Workshop, SFP'99*, pages 2–10. Intellect, 2000.

[5] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. *Patterns and Skeletons for Parallel and Distributed Computing. F. Rabhi and S. Gorlatch (eds.)*, chapter Parallelism Abstractions in Eden, pages 95–128. Springer-Verlag, 2002.

[6] I. Lynagh. Template Haskell: A report from the field. (http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/), 2003.

[7] I. Lynagh. Unrolling and Simplifying Expressions with Template Haskell. (http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/), 2003.

[8] I. Lynagh. Typing Template Haskell: Soft Types. (http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/), 2004.

[9] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Phd. thesis, technical report cst-15-81, Dept Computer Science, University of Edinburgh, December 1981.

[10] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[11] R. Peña and C. Segura. Sized Types for Typing Eden Skeletons. In T. Arts and M. Mohnen, editors, *Selected papers of the 13th International Workshop on Implementation of Functional Languages, IFL'01*, volume 2312 of *LNCS*, pages 1–17. Springer-Verlag, 2002.

[12] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele, DTI/SERC*, pages 249–257, 1993.

[13] S. L. Peyton Jones and W. Partain. Measuring the Effectiveness of a Simple Strictness Analyser. In *Glasgow Workshop on Functional Programming 1993*, Workshops in Computing, pages 201–220. Springer-Verlag, 1993.

[14] R. Peña and C. Segura. Non-determinism Analyses in a Parallel-Functional Language. *Journal of Functional Programming*, 15(1):67–100, 2005.

[15] S. Priebe. Preprocessing Eden with Template Haskell. In *Generative Programming and Component Engineering: 4th International Conference, GPCE 2005*, volume 3676 of *LNCS*, pages 357–372. Springer-Verlag, 2005.

[16] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Glasgow University, Dept. of Computing Science, 1995.

[17] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[18] S. Peyton Jones T. Sheard. Notes on Template Haskell Version 2. (http://research.microsoft.com/~simonpj/tmp/notes2.ps), 2003.

[19] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming. LNCS 925.* Springer-Verlag, 1995.

[20] P. L. Wadler and R. J. M. Hughes. Projections for Strictness Analysis. In G. Kahn, editor, *Proceedings of Conference Functional Programming Languages and Computer Architecture, FPCA'87*, volume 274 of *LNCS*, pages 385–407. Springer-Verlag, 1987.

[21] K. Wansbrough and S. L. Peyton Jones. Once Upon a Polymorphic Type. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–28, San Antonio, Texas, January 1999. ACM Press.