

# Strictness Analysis and let-to-case Transformation using Template Haskell\*

Carmen Torrano      Clara Segura

Departamento de Sistemas Informáticos y Programación  
Universidad Complutense de Madrid, Spain  
e-mail: [ctorranog@hotmail.com](mailto:ctorranog@hotmail.com), [csegura@sip.ucm.es](mailto:csegura@sip.ucm.es)

## Abstract

Metaprogramming consists of writing programs that generate or manipulate other programs. Template Haskell is a very 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.

## 1 INTRODUCTION

Metaprogramming consists of writing programs that generate or manipulate other programs. Template Haskell [16] is a very recent extension of Haskell, currently implemented in the Glasgow Haskell Compiler [11] (GHC), giving support to metaprogramming at compile time. 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 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 [13, 10, 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 does not seem much reasonable. Using Template Haskell any new analysis or transformation could be proved easily and incorporated to the compilation process without directly modifying the internals of the compiler.

---

\*Work partially supported by the Spanish project TIN2004-07943-C04.

```

data Exp =
  LitE Lit           -- literal
  VarE String       -- variable
  ConE String       -- constructor
  LamE [Pat] Exp    -- lambda abstraction
  AppE Exp Exp      -- application
  CondeE 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 String       -- variable
  ConP String [Pat] -- constructor
  . . .
data Body =
  NormalB Exp -- just an expression
  . . .
data Dec =
  ValD Pat Body [Dec] -- value
  FunD String [Clause [Pat] Body Dec] -- function
  . . .

```

**FIGURE 1. Data types representing Haskell syntax**

In this paper we explore the possibilities of Template Haskell 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.

## 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 syntactic categories of a Haskell program, like declarations (`Dec`) or patterns (`Pat`). In Figure 1 we show parts of the definitions of these data types, which we will use later in Section 4.

Once Template Haskell has processed a Haskell program, its abstract syntax tree is accessible and consequently transformations and analyses can easily be defined by cases over the syntax.

A quasi-quotation mechanism allowing to represent templates, i.e. Haskell programs at compile time, is available. Quasi-quotations are constructed by placing brackets, `[ |` and `| ]`, around concrete Haskell syntax fragments. This mechanism is built on top of a quotation monad `Q`:

```
instance Monad Q
runQ :: Q a -> IO a
. . .
type ExpQ = Q Exp
. . .
```

Obviously, as `Q` is a monad, the usual operators `bind`, `return` and `fail` are available, as well as the `do`-notation. Writing `[ | e | ]` "lifts" the Haskell expression `e` of type `Exp` into the monadic world, getting the type `ExpQ`.

Code can also be built using some functions available in the library `Language.Haskell.TH.Syntax`. For example, we can build expression `[ | \ x -> x | ]` also by writing `lam [pvar "x"] (var "x")`, where `lam`, `pvar` and `var` are library functions that build, respectively, lambda abstractions, pattern variables and expression variables.

Once the expression "lives" in the monadic world, parts of it can be evaluated at compile time by means of the splice notation `$`. This means compile-time evaluation when placed at top level or inside a quasi-quoted expression. The result of the 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' | ]`.

There are other features of Template Haskell we are not using here; the interested reader may look at [16] for more details.

### 3 STRICTNESS ANALYSIS AND LET-TO-CASE TRANSFORMATION

Lazy 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.

The strictness analysis [8, 1, 17, 2] detects those 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

$e$	$\rightarrow$	$c$	{ constant }
		$v$	{ variable }
		$e_1 \text{ op } e_2$	{ primitive operator }
		$\text{if } e_1 \text{ then } e_2 \text{ then } e_3$	{ conditional }
		$\lambda b. e$	{ first-order lambda }
		$C e_1 \dots e_n$	{ constructor application }
		$e_1 e_2$	{ function application }
		<b>let</b> $v_1 = e_1 \dots v_n = e_n$ <b>in</b> $e$	{ let expression }
		<b>case</b> $e$ <b>of</b> $alt_1 \dots alt_n$	{ case expression }
$alt$	$\rightarrow$	$C b_1 \dots b_n \rightarrow e$	
		$b \rightarrow e$	

**FIGURE 2. A first-order subset of Haskell**

be immediately evaluated, so that the let expression can be transformed into a case expression without modifying the expression semantics [15]. This is known as *let-to-case* transformation:

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

Strictness analysis can be done by using abstract interpretation [9]. 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. In [8] Mycroft gave for the first time an abstract interpretation based strictness analysis for a first-order functional language. Later, Burn et al. [1] extended it to higher order programs and Wadler [17] introduced the analysis of data types. In [12] Peyton Jones and Par-tain described how to use signatures in order to make abstract interpretation more efficient.

We show here an abstract interpretation based strictness analysis for a first-order functional language with data types. In the next section we describe how we have implemented it by using signatures to represent functions in an efficient way. For technical reasons, in Section 5 we will explain why, the language being analysed is a first-order subset of Haskell with data types, whose syntax is shown in Figure 2.

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 basic or an algebraic value. 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. If we lifted the previously mentioned restrictions we would have a higher-order subset of Haskell.

$$\begin{aligned}
\llbracket c \rrbracket \rho &= \top \\
\llbracket v \rrbracket \rho &= \rho(v) \\
\llbracket e_1 \text{ op } e_2 \rrbracket \rho &= \llbracket e_1 \rrbracket \rho \sqcap \llbracket e_2 \rrbracket \rho \\
\llbracket \text{if } e_1 \text{ then } e_2 \text{ then } e_3 \rrbracket \rho &= \llbracket e_1 \rrbracket \rho \sqcap (\llbracket e_2 \rrbracket \rho \sqcup \llbracket e_3 \rrbracket \rho) \\
\llbracket \lambda b.e \rrbracket \rho &= \lambda a. \llbracket e \rrbracket (\rho + [b \rightarrow a]) \\
\llbracket C e_1 \dots e_n \rrbracket \rho &= \llbracket e_1 \rrbracket \rho \sqcup \dots \sqcup \llbracket e_n \rrbracket \rho \\
\llbracket e_1 e_2 \rrbracket \rho &= \llbracket e_1 \rrbracket \rho \llbracket e_2 \rrbracket \rho \\
\llbracket \text{let } v_1 = e_1 \dots v_n = e_n \text{ in } e \rrbracket \rho &= \llbracket e \rrbracket \rho' \\
&\quad \text{where } \rho' = \text{fix } f \\
&\quad \quad f = \lambda \rho. \rho + [v_1 \rightarrow \llbracket e_1 \rrbracket \rho, \dots, v_n \rightarrow \llbracket e_n \rrbracket \rho] \\
\llbracket \text{case } e \text{ of } alt_1 \dots alt_n \rrbracket \rho &= a \sqcap (a_1 \sqcup \dots \sqcup a_n) \\
&\quad \text{where } a = \llbracket e \rrbracket \rho \\
&\quad \quad a_i = \llbracket alt_i \rrbracket \rho a \\
\llbracket C b_1 \dots b_n \rightarrow e \rrbracket \rho a &= \llbracket e \rrbracket (\rho + [b_1 \rightarrow a, \dots, b_n \rightarrow a]) \\
\llbracket b \rightarrow e \rrbracket \rho a &= \llbracket e \rrbracket (\rho + [b \rightarrow a])
\end{aligned}$$

**FIGURE 3. A strictness analysis by abstract interpretation**

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 3 we show the interpretation of each of the language expressions, where  $\rho$  represents an abstract environment assigning abstract values to variables. The environment  $\rho + [v \rightarrow av]$  either extends environment  $\rho$  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.

Primitive binary operators, like  $+$  or  $*$ , are strict in both arguments so we use  $\sqcap$  operator. The abstract value of a constructor application is obtained by collapsing the abstract values of its components into a single basic abstract value. We use  $\sqcup$  in this case so that  $\perp$  represents a completely undefined constructed value. The reason for this is laziness of the constructors. This means for example, that function  $\lambda x.x : []$  is not considered strict in its first argument. Notice that by using  $\sqcup$  we lose information about the components. In a case expression we need to recover it in order to give abstract values to the bound variables of the alternatives. Only if the abstract value of the discriminant is  $\perp$  we can be sure that all the components had  $\perp$  as abstract value, otherwise "we don't know". So the variables bound by the case alternatives inherit the abstract value of the discriminant.

As we have used first-order abstract functions as abstract values, function ap-

plication can be easily interpreted as abstract function application. To interpret a let expression we need a standard fixpoint calculation as it may be recursive.

#### 4 IMPLEMENTATION USING TEMPLATE HASKELL

In this section we describe the implementation of the strictness analysis and the corresponding let-to-case transformation using Template Haskell.

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 `i`th argument [12]. The special `FB n` value is the abstract value of a completely undefined function with `n` arguments, that is, the bottom of the functional abstract domain, which is useful in several places.

The analysis receives a closed expression written in Haskell that belongs to the previously mentioned subset and an initial empty strictness environment, and returns its abstract value. We want the analysis to happen at compile time, so we need the mechanisms described in Section 2. The quasi-quotation notation allows us to write programs as input data for our analysis very easily, we just need to write the program in Haskell and then lift it to the monadic world. For example:

```
q :: ExpQ
q = [| \ x -> \ y -> 3 * x |]
```

The analysis is defined as a transformation from a Haskell expression to an abstract value. As we have seen, the expression we receive belongs to the monadic world, so such transformation will take place inside it:

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

Function `toExp :: AbsVal -> Exp` just converts an abstract value into an expression so that the result of the analysis also lives in the monadic world. Function `strict` is the actual strictness analysis defined by cases over the syntax, we just need to know the `Exp` data type definition (shown in Figure 1) and the restrictions of our language (explained in the previous section).

Finally we use `$` in order to execute `strict2` at compile time:

```
main = putStr (show $(strict2 q empty))
```

Here `empty` represents the empty strictness environment.

```

strict :: Exp -> Env -> AbsVal
strict (LitE l) rho = B Top
strict (InfixE (Just e1) e (Just e2)) rho =
  inf (strict e1 rho) (strict e2 rho)
strict (VarE s) rho = getEnv s rho
strict (CondeE e1 e2 e3) rho =
  inf (strict e1 rho)
    (sup (strict e2 rho) (strict e3 rho))

```

**FIGURE 4. Implementation of the strictness analysis (I)**

```

strict (LamE ((VarP s):[]) e) rho =
  let B b = strictaux e (addEnv (s,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

```

**FIGURE 5. Implementation of the strictness analysis (II)**

Now we describe in more detail the definition of function `strict`, which makes the abstract interpretation. In Figure 4 we show the interpretation of constants, primitive operators, variables and conditional expressions, as shown in the previous section. There, `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 5 we show the interpretation of a lambda abstraction. Its value is a signature  $F [b_1, \dots, b_n]$ , being  $n$  the number of arguments, obtained by probing the function with several combination of arguments. The component  $b_i$  is obtained by applying the function to `B Top` in each argument position except the  $i$ th one, where the function is applied to `B Bot`. For example, if the function has three arguments the probing is made with the following combinations of arguments:  $(B Bot, B Top, B Top)$ ;  $(B Top, B Bot, B Top)$  and  $(B Top, B Top, B Bot)$ .

We do not know how many arguments the function has, so we start probing the function with the first argument. First, we give it the value `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 6 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,

```

strict (ConE cons) rho = B Top
strict (AppE (ConE cons) e) rho = strict e rho
strict (AppE e1 e2) rho =
  if (isCon e1) then sup (strict e1 rho) (strict e2 rho)
  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 xs) (B b) =
  let h = head xs in
  let tl = tail xs in
  if (h == Top || b == Top) then
    if isempty(tl) then (B Top)
    else F tl
  else
    if isempty(tl) then B Bot
    else (FB (length tl))

```

**FIGURE 6. Implementation of the strictness analysis (III)**

so we use function `isCon` to distinguish them. If it is a constructor application we apply the least upper bound operator.

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 `F xs'` where `xs'` has one element less than `xs`. 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` then the function is strict in its first argument and that is undefined so we can return `B Bot` independently of the rest of the arguments. So, if there are arguments left we return the completely undefined function `FB (n-1)`.

In Figure 7 we show the interpretation of a let expression. Auxiliary function `strictdecs` carries out the fixpoint calculation. The initial environment `init` is built by extending the environment with the new variables bound to an undefined abstract value of the appropriate type. Function `combines` updates the environ-

```

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',cont) ->
      let
        aes = map (flip strict rho') es
        pairs = zipWith pair varns aes
      in
        (combines rho' pairs)
    fix g (env,b) =
      if b then
        ( let (env',b') = g (env,b)
          in (fix g (env',b')))
      else env
  in
    fix f (init,True)

```

**FIGURE 7. Implementation of the strictness analysis (IV)**

```

strict (CaseE e ms) rho =
  let se = strict e rho
      l = caseaux ms se rho
  in (inf se (suplist l))

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 (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'

```

**FIGURE 8. Auxiliary functions for case and let expressions**

ment 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 8 we show the interpretation of a case expression. 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`.

Notice that there are several places where it would have been useful to know the type of an expression: When probing functions, when distinguishing function and constructor applications and when building the `init` environment. This is even more important if we want the analysis to be higher-order because the abstract values used to probe a function may be functional as well.

The analysis returns `F [Bot, Top]`, as expected, for the previous example `q`, i.e. the function is strict in the first argument but not in the second one. Another example with a case expression is the following one:

```
r = [| \ 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.

However, 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 -> C x y` has abstract value  $\lambda a_1.\lambda a_2.a_1 \sqcup a_2$  but the implementation would assign it signature `F [Top, Top]` which is undistinguishable from abstract value  $\lambda a_1.\lambda a_2.\top$ . Function `\ x -> \ y -> 1` would have the same signature.

The let-to-case transformation has been developed in a similar way, but here we have used the `Q` monad in order to view the result of the transformation. Otherwise we would just view the result of the evaluation of the expression. We have used the function `runQ` of the monad to obtain a printable result:

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

where `transf2` is defined in a similar way to `strict2`.

The function making all the important work is `transf`. We show in Figure 9 only the most interesting case, the let expression. We are assuming that when there are several definitions together in a let expression is because they are mutually recursive. The desugarer usually partitions these definitions into strongly connected components, so we should be able to get benefit from it (see Section 5).

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 body of the let. In that case the abstract values of the

```

transf :: Exp -> Env -> Exp
transf (LetE ds e) rho =
  if (isRecorFun ds) then
    let (vs,es) = splitDecs ds
        rho' = foldr addEnvtop rho vs
        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
            CaseE te' [Match (VarP x) (NormalB te) []]
          else
            LetE ds' te

```

**FIGURE 9. Transformation of a let expression**

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. We have tried several examples like the following one:

```
[ | let a = 1 in let b = 2 in a + b | ]
```

It is transformed into:

```

CaseE (LitE (IntegerL 1))
[Match (VarP "a'1")
(NormalB (CaseE (LitE (IntegerL 2))
[Match (VarP "b'0")
(NormalB (InfixE (Just (VarE "a'1"))
(VarE "GHC.Num:+")
(Just (VarE "b'0"))))) [ ])) [ ]]

```

which represents the expression **case 1 of**  $a \rightarrow$  **case 2 of**  $b \rightarrow a + b$ .

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:

```
[ | \ x -> let a = 1 in a + x | ]
```

is transformed into:

```
LamE [VarP "x'0"]
(CaseE (LitE (IntegerL 1))
[Match (VarP "a'1")
(NormalB (InfixE (Just (VarE "a'1"))
                (VarE "GHC.Num:+")
                (Just (VarE "x'0")))) [ ]])
```

which represents the expression  $\lambda x. \text{case } a \text{ of } 1 \rightarrow a + x$ .

## 5 RELATED AND FUTURE WORK

Template Haskell is a very recent extension of Haskell for metaprogramming, currently implemented in GHC 6.0. In [16] the design of the extension and the facilities it offers are described in detail. Its functionality is obtained from the library `Language.Haskell.Syntax`. Template Haskell has revealed as 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 [14] without modifying GHC.

We have presented the implementation of a strictness analysis and a subsequent let-to-case transformation using Template Haskell. These are well-known problems, which has allowed us to concentrate on the difficulties and limitations of using Template Haskell for our purposes. As far as we know, this is the first time that Template Haskell has been used for developing a static analysis. The analysis has been done for a first-order subset of Haskell. This has been relatively easy to define. We just needed to know how to get inside and outside the monadic world where the compile time computations take place, and then we can work by cases over the syntax constructions forgetting about the monad. The only difficulty here is the absence of a properly commented documentation about the library. We have used version 0.6 of the library, but intend to benefit from later versions that could help to improve the design of the analysis.

The analysis could be extended to higher-order programs using nested signatures. We have not done this for the moment for two reasons. First, there are some aspects of the tool that are not implemented yet and that would help to obtain an elegant version of the higher-order analysis. When analysing higher order functions, it is necessary to generate the appropriate signatures in correspondence with their types. Although there is a typing algorithm for Template Haskell, we are not able to observe the type information kept in the abstract tree. We could of course do 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.

Second, analyses and transformations are usually done over a simplified language where the syntactic sugar has disappeared (Core in GHC), so it does not seem necessary to develop an analysis for the full language Haskell. Of course this had 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 but we think it would be very useful. In particular, in Section 4 we have seen that we could benefit from the dependency analysis done by the compiler. 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, because in that case we want to reference the original variables written by him/her, which are usually lost in further phases of the compiler.

## 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] 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.
- [9] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- [10] 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.
- [11] 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.
- [12] 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.

- [13] R. Peña and C. Segura. Non-determinism Analyses in a Parallel-Functional Language. *Journal of Functional Programming*, 15(1):67–100, 2005.
- [14] S. Priebe. A Framework for Enhancing Eden Code with Template Haskell. In *Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL'04. Technical Report 0408, Christian-Albrechts-Universität zu Kiel.*, pages 455–456, 2004.
- [15] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Dept. of Computing Science, 1995.
- [16] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.
- [17] 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.