Optimising Embedded DSLs using Template Haskell

Sean Seefried, Manuel Chakravarty and Gabriele Keller

sseefried@cse.unsw.edu.au

October 27, 2004

① EDSLs

② The approach

③ Template Haskell

④ Pan $\longrightarrow$ PanTheon

⑤ Evaluation of TH

EDSLs

An Embedded Domain Specific Language is essentially just a library writte in a rich "host" language.

Features which help:

- ➜ Higher order functions
- ➜ Syntax extension mechanisms
- ➜ Flexible/extensible type system

Advantages:

- Easier to write
  - No need to write lexer, parser, type checker, code generator etc
- Easier to distribute

Disadvantages:

- The code generated is slow
  - no optimisation at level of DSL
- Error messages relate to host language, not DSL
- Poor tool support e.g. profilers, debuggers

Advantages:

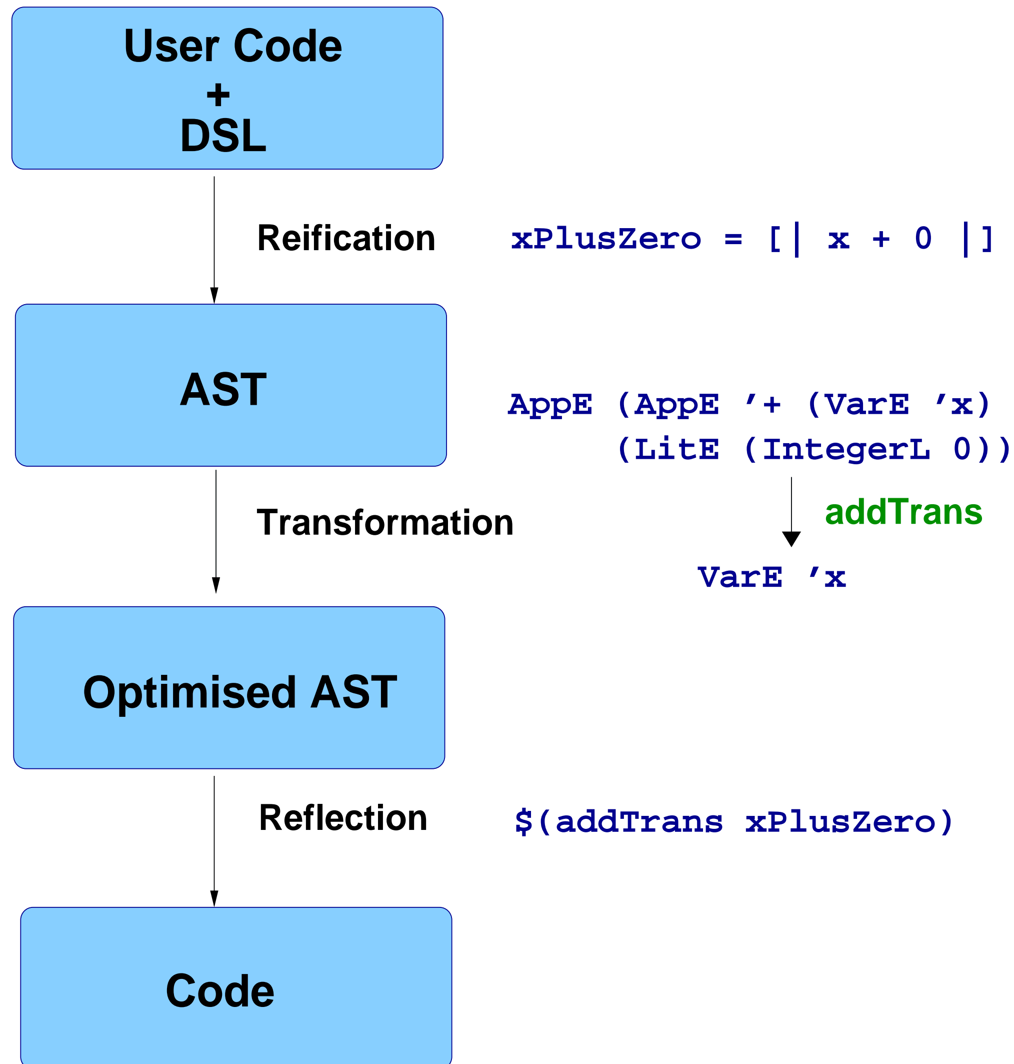
- Easier to write
  - No need to write lexer, parser, type checker, code generator etc
- Easier to distribute

Disadvantages:

- The code generated is slow
  - no optimisation at level of DSL
- Error messages relate to host language, not DSL
- Poor tool support e.g. profilers, debuggers

Compile-time meta-programming improves efficiency

THE APPROACH: SOURCE TRANSFORMATION



TEMPLATE HASKELL

Designed by Tim Sheard and Simon Peyton Jones.

Features:

- ➜ compile-time meta-programming
- ➜ programs represented as ordinary algebraic data types
- ➜ interesting approach to type checking:
  - ➜ deferred until all *splicing* done
  - ➜ in contrast to MetaML which types meta-programs statically
- ➜ Source transformation handled natively as a *language feature*
  - Quasi-quote checks syntax and respects scope

Where are we in the talk?

① EDSLs

② The approach

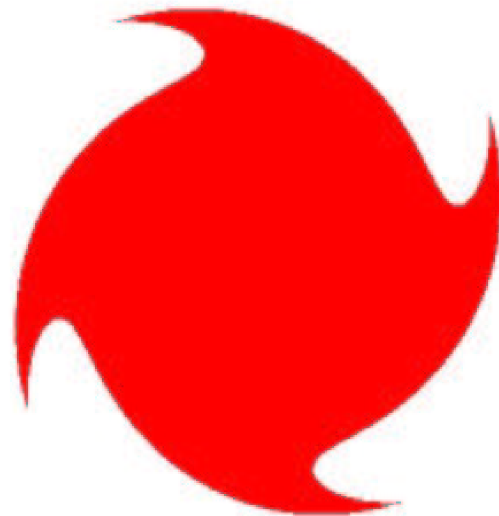
③ Template Haskell

④ Pan → PanTheon

⑤ Evaluation of TH

PAN: A 2D IMAGE/ANIMATION SYNTHESIS LANGUAGE

$$\begin{aligned} square\ s\ (x, y) &= if\ -s/2 < x\ \&\&\ x < s/2 \\ &\quad -s/2 < y\ \&\&\ y < s/2 \\ &\quad then\ red\ else\ invisible \\ warped_square &= swirl\ 75.0\ (square\ 50) \end{aligned}$$



`warped_square`

Previously implemented by Elliott, de Moor, Finne.

An embedded *compiler*:

- ➜ User of Pan is actually constructing ASTs
- ➜ back-end of compiler needs to be written
- ➜ But front-end doesn't

Previously implemented by Elliott, de Moor, Finne.

An embedded *compiler*:

- ➜ User of Pan is actually constructing ASTs
- ➜ back-end of compiler needs to be written
- ➜ But front-end doesn't

Disadvantages:

- ➜ Native optimisations of host lang. compiler
- ➜ It's still a lot of work: 13000 lines of Haskell

PanTHeon

PanTHeon is an implementation of Pan in Template Haskell (TH).

The approach:

- ➜ Retains sharing of host lang. features (as per EDSLs)
- ➜ improves performance through *domain specific optimisations* implemented using meta-programming techniques

Retains host language optimisations

Could also be thought of as an *Active Library*

PanTHeon

PanTHeon is an implementation of Pan in Template Haskell (TH).

The approach:

- Retains sharing of host lang. features (as per EDSLs)
- improves performance through *domain specific optimisations* implemented using meta-programming techniques

Retains host language optimisations

Could also be thought of as an *Active Library*

Three main optimisations applied:

- algebraic transformation
- aggressive inlining
- unboxing of arithmetic

Algebraic Transformation

$$
\begin{aligned}
&empty \text{ `}over\text{` } image &&= image\\
&image \text{ `}over\text{` } image &&= image\\
&fromPolar\ (toPolar\ f) &&= f\\
&toPolar\ (fromPolar\ f) &&= f\\
&rotate\ a1\ (rotate\ a2\ image) &&= rotate\ (a1\ +\ a2)\ image\\
&scale\ (x1, y1)\ (scale\ (x2, y2)\ image) &&= scale\ (x1 * x2,\ y1 * y2)\ image\\
&distO\ (fromPolar\ (r, a)) &&= r
\end{aligned}
$$

Easy with *pattern matching*:

$$
\begin{aligned}
&algTrans\ exp@(AppE\ (AppE\ (VarE\ 'over)\ image)\ image')\\
&\mid image\ ==\ 'image\ =\ algTrans\ image\\
&\mid otherwise\ =\ \ldots
\end{aligned}
$$

INLINING

- ➜ Equivalent λ expression created for function body
- ➜ Performed to fixed depth (so that recursion isn't an issue).
- ➜ Code replication mitigated by GHC's C.S.E. pass

as well soon see...

$$distO\ (x, y) = sqrt\ (x * x + y * y)$$
$$rotate\ a\ (x, y) = (x * cos\ a - y * sin\ a,\ y * cos\ a + x * sin\ a)$$
$$swirlP\ r = \lambda\ p \rightarrow rotate\ (distO\ p * (2 * pi/r))\ p$$

to

$$(\lambda\ (x, y) \rightarrow$$
$$(x * cos\ (sqrt\ (x * x + y * y) * (2 * pi/r)) -$$
$$y * sin\ (sqrt\ (x * x + y * y) * (2 * pi/r)),$$
$$y * cos\ (sqrt\ (x * x + y * y) * (2 * pi/r))$$
$$+ x * sin\ (sqrt\ (x * x + y * y) * (2 * pi/r)))$$

... back to

$$
\begin{aligned}
&\lambda\,(x, y) \;\rightarrow \\
&let\ sqrt' \;=\; sqrt\,(x * x \;+\; y * y) * (6.283185r) \\
&\quad\ \ cos' \;=\; cos\ sqrt' \\
&\quad\ \ sin' \;=\; sin\ sqrt' \\
&in\,(x \;*\; cos' \;-\; y \;*\; sin',\; y * cos' \;+\; x * sin')
\end{aligned}
$$

UNBOXING

Why do it?:

- ➜ Less indirection
- ➜ Better memory locality

Boxed:

$$
\begin{array}{l}
type\ Colour\ =\ (Float,\ Float,\ Float,\ Float)\\
type\ Point\ =\ (Float,\ Float)\\
lerpC\ w\ (r1, g1, b1, a1)\ (r2, g2, b2, a2)\ =\\
(h\ r1\ r2,\ h\ g1\ g2,\ h\ b1\ b2,\ h\ a1\ a2)\\
\quad \textbf{where}\\
\qquad h\ x1\ x2\ =\ w\ *\ x1\ +\ (1\ -\ w)\ *\ x2
\end{array}
$$

To unboxed:

$$
\begin{array}{l}
data\ Colour_UB \ = \ Colour_UB\ Float\#\ Float\#\ Float\#\ Float\# \\
data\ Point_UB \ = \ Point_UB\ Float\#\ Float\# \\
lerpC\ (Colour_UB\ r1\ g1\ b1\ a1)\ (Colour_UB\ r2\ g2\ b2\ a2) \ = \\
Colour_UB\ (h1\ r1\ r2)\ (h\ g1\ g2)\ (h\ b1\ b2)\ (h\ a1\ a2) \\
\quad \mathbf{where} \\
\qquad h\ x1\ x2 \ = \ (w\ `timesFloat\#`\ x1)\ `plusFloat\#` \\
\qquad (1.0\#\ `minusFloat\#`\ w)\ `timesFloat\#`\ x2
\end{array}
$$

Unboxing is problematic

Restrictions:

- ➜ Can't be contained within polymorphic data structures
- ➜ Can't be passed to polymorphic functions

With type information we could:

- ➜ Specialise polymorphic functions
- ➜ Specialise polymorphic data structures
- ➜ This amounts to *full monomorphisation*

Unboxing is problematic

Restrictions:

- ➜ Can't be contained within polymorphic data structures
- ➜ Can't be passed to polymorphic functions

With type information we could:

- ➜ Specialise polymorphic functions
- ➜ Specialise polymorphic data structures
- ➜ This amounts to *full monomorphisation*

But we have no types!

More on types

Correct/complete transformation is impossible without types

Consider this:

- What is the type of a literal?
- What about overloaded functions?
- How can we specialise without types?

Proposal: Closed expressions should be typed

RESULTS

In 4000 lines of Haskell code we have something of almost equivalent functionality with acceptable performance.

Effect	Base	Inlined	Unboxed	Both
checker_swirl	**8.86 f/s**	1.309x	2.190x	2.258x
circle	**11.241 f/s**	1.324x	2.126x	2.083x
checker_on_stripes	**1.302 f/s**	1.027x	8.361x	9.003x
four_squares	**2.512 f/s**	1.366x	4.184x	4.152x
triball	**1.244 f/s**	1.914x	2.578x	2.707x
tunnel_view	**4.62 f/s**	1.223x	2.042x	2.661x

An evaluation of Template Haskell

What it does well:

- ➜ Native means to perform source transformation
- ➜ Transformations easy to write using pattern matching and generic programming techniques (Scrap your Boilerplate)

Better as a program *generator* than *transformer*

What could be better:

- ➜ Laziness by default. Hard to strictify
- ➜ All TH optimisations before those of compiler
- ➜ Syntax too rich? Template Core?
- ➜ Needs a means of reifying entire modules
- ➜ Should provide types for closed expressions/declarations

An alternative

Use a light weight parser and a novel compilation framework:

- ➜ Light weight parser solves syntax problem
- ➜ Use a compilation framework in which one can plug in custom optimisations passes
- ➜ Should be over an intermediate representation
- ➜ Order of optimisations should be specifiable
- ➜ Staging capabilities would still be useful

The End