Slide 1

EDSLs, Meta-programming and Devil

Sean Seefried

sseefried@cse.unsw.edu.au

May 18, 2004

Slide 2

A Parser that looks like a Grammar

$expr$	::=	$term{+}term \mid term$
$term$	::=	$factor{*}factor \mid factor$
$factor$	::=	$digit^{+} \mid (expr)$
$digit$	::=	0|1|..|9

```
expr   = (term `cat` str "+" `cat` term) `alt` term
term   = (factor `cat` str "*" `cat` factor) `alt` factor
factor = some digit `alt` (str "(" `cat` expr `cat` str ")")
digit  = lit '0' `alt` ... `alt` lit '9'
```


Slide 3

A Domain Specific Language for Parsers

```
type Parser a = String → [(a, String)]
```

In the words of Dr Seuss:

A Parser for Things,
is a Function from Strings,
to Lists of Pairs,
Of Things and Strings!

Slide 4

Primitive parsers

```
fail             :: Parser α
fail inp         = []
succeed          :: α → Parser α
succeed v inp    = [(v, inp)]
lit              :: Char → Parser Char
lit c []         = fail []
lit c (x : xs)   | c == x = succeed x xs
                 | otherwise = fail xs
```

Slide 5

Parsers from Parsers

Concatenation and alternation:

$$
\begin{aligned}
&alt && :: Parser\ \alpha \rightarrow Parser\ a \rightarrow Parser\ \alpha \\
&(p1\ `alt`\ p2)\ inp && = p1\ inp\ +\!+\ p2\ inp \\
&cat && :: Parser\ \alpha \rightarrow Parser\ \beta \rightarrow Parser\ (\alpha, \beta) \\
&(p1\ `cat`\ p2)\ inp && = [((v1, v2),\ out2) \mid (v1,\ out1) \leftarrow p1\ inp, \\
& && \qquad\qquad\qquad\qquad\quad (v2,\ out2) \leftarrow p2\ out1]
\end{aligned}
$$

$(\ldots)^*$ and $(\ldots)^+$, or Star and Plus:

$$
\begin{aligned}
&many && :: Parser\ \alpha \rightarrow Parser\ [\alpha] \\
&many\ p && = ((p\ `cat`\ many\ p)\ `using`\ cons)\ `alt`\ succeed\ [] \\
&some && :: Parser\ \alpha \rightarrow Parser\ [\alpha] \\
&some\ p && = (p\ `cat`\ many\ p)\ `using`\ cons
\end{aligned}
$$

Slide 6

Structure of talk

A rough outline:

- Domain Specific Languages
  - What are they?
  - Why use them?
- Embedded DSLs
  - What does *embedded* mean?
  - How they foster reuse
  - Why they're *not* used
- Meta-programming
  - What is it?
  - How do we apply it to EDSLs?
- Devil - A DEVice Interface Language

Slide 7

Domain Specific Languages

Why use them?:

- Reduced programming effort
  - Fewer lines of code
  - Reduced likelihood of errors
  - Programs easier to reason about
- Not feasible to keep adding features to general purpose languages
  - Syntax of GP language limited
  - Only of interest to minorities
- Can be used by non-programmers!
  - Already familiar with *domain* and *notation*

But creating new languages is expensive

Slide 8

Language Creation

Is expensive because:

- Many components to implement. e.g. lexer, parser, code generator
  - Such components can be quite similar for many languages
- A programming language is not just the compiler
  - Tools. e.g. debuggers, parsers, IDEs
  - standards documents

Slide 9

EDSLs

Embedded Domain Specific Languages are one solution

Why not write *languages* as *libraries* in a rich host language?

Good host language features:

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

Slide 10

Inheritance from host language: No need to write:

- ➜ Lexer
- ➜ Parser
- ➜ Type Checker
- ➜ Optimisation phases
- ➜ Code generator

Slide 11

DISADVANTAGES OF EDSLS

EDLSs have the following problems:

- ➜ The code generated can be slow:
  - ➜ No optimisation for DSL terms
  - ➜ Host language cannot *see* at level of DSL
- ➜ Error messages in terms of host language not DSL
- ➜ Tools only support host language
  - ➜ profilers
  - ➜ debuggers

The first of these problems can be solved via meta-programming

Slide 12

WHAT'S NEXT?

- ➜ What is meta-programming?
- ➜ Optimisation via meta-programming
- ➜ Devil
- ➜ Anti-climax. Why MP isn't applicable to Devil.
- ➜ Introduction to Template Haskell
- ➜ Optimisation using Template Haskell

Slide 13

What is meta-programming?

Meta-programs are those that *analyse*, *generate* or *transform* others.

Two core concepts:

- ➜ Reification. Code is turned into data
- ➜ Reflection. Also known as splicing. Data is turned back into code.

Good language natively deals with:

- ➜ syntax. Rejects incorrect programs.
- ➜ abstraction mechanisms
- ➜ binding and scoping

Slide 14

Optimisation via meta-programming

Transform user-written code to semantically equivalent but *faster* code

- ➜ We *teach* the compiler about the EDSL
  - ➜ EDSL is likely to have algebraic properties
  - ➜ Can create efficient representations
- ➜ Host language optimisations *inherited*
  - ➜ Don't have to rewrite a lot of tedious well-known optimisations
  - ➜ *Drawback.* You can't intersperse them with user-written ones

We'll return to this later. But now ... Devil

Slide 15

Devil

DEVice Interface Language:

- ➜ Language for specification of API for hardware devices
- ➜ Automatically generates low-level code
- ➜ Three levels of abstraction:
  - ➜ port
  - ➜ register
  - ➜ variable
- ➜ Does various consistency checks

Advantages:

- ➜ abstraction level raised. Aids
  - ➜ debugging
  - ➜ maintenance
- ➜ reduces likelihood of low-level errors

Slide 16

Comparison of Devil specifications:

```
device logitech_busmouse (base : bit[8] port @ {0..3})
{
  // Index register
  register index_reg = write base @ 2, mask '1..00000' : bit[8];
  private variable index = index_reg[6..5] : int(2);
  register x_low = read base @ 0, mask '****....' : bit[8];
  register x_high = read base @ 1, mask '****....' : bit[8];
  variable dx = x_high[3..0] # x_low[3..0] : signed int(8);
}
logitech_busmouse =
 Device [DeviceParam "base" (ArrayType 8 BitType) [Ascending 0 3]]
 [
   ObjReg (Reg "index_reg" [RegPort Write ("base" :@: 2) ] []
            (ArrayType 8 BitType))
  , ObjVar (Var "index" True
                  [VarRegBitsRange "index_reg" (Descending 6 5)]
                  (IntType False 2))
  , ObjReg (Reg "x_low" [RegPort Read ("base" :@: 0)]
             [Mask [Star, Star, Star, Star, Dot, Dot, Dot, Dot]]
             (ArrayType 8 BitType))
  , ObjReg (Reg "x_high" [RegPort Read ("base" :@: 1)]
             [Mask [Star, Star, Star, Star, Dot, Dot, Dot, Dot]]
             (ArrayType 8 BitType))
  , ObjVar (Var "dx" False
             [((VarRegBitsRange "x_high" (Descending 3 0)) :#:
              (VarRegBitsRange "x_low" (Descending 3 0)))]
             (IntType True 8))
 ]
```

Slide 17

ANTI-CLIMAX WHY MP ISN'T APPLICABLE TO DEVIL

We don't want to use GHC's code generator

- Why?
  - *Not* because we can't produce fast code
  - But because we don't want GHC's run-time
    - Makes binaries too large
    - Contains things we don't need
    - People will write their drivers in C anyway. (And may need to.)
- C is a good language for writing low-level operations
  - C is "close" to assembly code generated
  - Small binaries produced

Slide 18

DEVIL - AN EMBEDDED COMPILER

It's not all bad:

- Still inherit the front end. i.e. lexer, parser, type checker
- Many syntax errors still caught
- *But the laborious task of writing the back-end is still required*
  - Well-known optimisations have to be written again
  - Code must be generated
- Fortunately, code generation doesn't seem to be too difficult for Devil
  - Close match between specification and C code to be generated

Slide 19

INTRODUCTION TO TEMPLATE HASKELL

Features:

- Compile-time
- Represents programs as ordinary algebraic data type.
- Two methods of reification:
  - quasi-quote mechanism
  - construction via algebraic data type
- Allows splicing of *top-level* declarations not just expressions
- Novel approach to typing - deferred until program generation complete

Slide 20

Quasi-quotes and splicing:

$$\begin{aligned} &data\ Format\ = D \mid S \mid L\ String \\ &printf\ = gen\ (parse\ s)\ [\mid \text{""} \mid] \end{aligned}$$

$$\begin{aligned}
&gen && :: [Format] -> Q\ Exp -> Q\ Exp \\
&gen\ []\ exp && = exp \\
&gen\ (D : xs)\ exp && = [\mid \lambda\ n \rightarrow \$(gen\ xs\ [\mid \$(exp)\ ++\ show\ n \mid])\mid] \\
&gen\ (S : xs)\ exp && = [\mid \lambda\ s \rightarrow \$(gen\ xs\ [\mid \$(exp)\ ++\ s \mid])\mid] \\
&gen\ (L\ s : xs)\ exp && = gen\ xs\ [\mid \$(exp)\ ++\ s \mid]
\end{aligned}$$

Slide 21

Algebraic construction/manipulation:

$$
\begin{aligned}
\mathit{data}\ \mathit{Exp} &= \mathit{VarE}\ \mathit{Name} \mid \mathit{LitE}\ \mathit{Lit} \mid \mathit{AppE}\ \mathit{Exp}\ \mathit{Exp} \mid \\
&\quad \mathit{LetE}\ [\mathit{Dec}]\ \mathit{Exp} \mid \ldots \\
\mathit{data}\ \mathit{Dec} &= \mathit{FunD}\ \mathit{Name}\ [\mathit{Clause}] \mid \mathit{DataD}\ \mathit{Cxt}\ \mathit{Name}\ \ldots \mid \\
&\quad \mathit{SigD}\ \mathit{Name}\ \mathit{Type} \\
\mathit{data}\ \mathit{Lit} &= \mathit{IntegerL}\ \mathit{Integer} \mid \mathit{StringL}\ \mathit{String} \mid \ldots
\end{aligned}
$$

Slide 22

Algebraic construction/manipulation (cont.):

$$
\begin{array}{l}
[\mid \mathit{let}\ \mathit{inc}\ x = x + 1\ \mathit{in}\ \mathit{inc}\ 3 \mid] \\
\equiv \\
\mathit{LetE}\ [\mathit{FunD}\ \ \mathit{inc}_0 \\
\qquad\qquad [\mathit{Clause}\ \ [\mathit{VarP}\ x_1] \\
\qquad\qquad\qquad (\mathit{NormalB}\ (\mathit{InfixE} \\
\qquad\qquad\qquad\qquad\qquad (\mathit{Just}\ (\mathit{VarE}\ x_1)) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{VarE}\ \mathit{GHC.Num.}{+}) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{Just}\ (\mathit{LitE}\ (\mathit{IntegerL}\ 1))))) \\
\qquad\qquad\qquad []]] \\
\qquad (\mathit{AppE}\ (\mathit{VarE}\ \mathit{inc}_0)\ (\mathit{LitE}\ (\mathit{IntegerL}\ 3)))
\end{array}
$$

Slide 23

OPTIMISATION EXAMPLE

- Parser combinators are equivalent to recursive descent parsers
- Well-known that these don't work for left-recursive grammars
- This can be fixed very simply by doing *left factoring*
- But is this really something we want the user to worry about?

Use meta-programming to do it!

Left factoring simply exploits the following equivalence:

$\alpha ::= \alpha\beta|\gamma$ is equivalent to $\alpha ::= \gamma\beta^*$

This is an example of *algebraic transformation*.

Slide 24

Consider a language containing left associative application

$$
\begin{aligned}
\mathit{expr} &= (\mathit{expr}\ \text{`}\mathit{cat}'\text{`}\ \mathit{prim})\ \text{`}\mathit{alt}\text{`}\ \mathit{prim} \\
\mathit{prim} &= \mathit{var}\ \text{`}\mathit{alt}\text{`}\ \mathit{num}
\end{aligned}
$$

Leaving aside implementation details we could expect TH to rewrite this as:

$$\mathit{expr} = \mathit{prim}\ \text{`}\mathit{alt}\text{`}\ (\mathit{many}\ \mathit{expr})$$

Slide 25

The End