Teaching the Construction of Domain Specific Languages

Pieter Koopman Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
pieter@cs.ru.nl rinus@cs.ru.nl

The target of our course advanced functional programming is currently shifting towards the design, use, and the functional implementation techniques for embedded Domains Specific Languages, DSLs. We use a well-known and very simple imperative language as running example to demonstrate the similarities and differences between shallow and deep embeddings of such a language. In this paper we briefly show how this is done and indicate some of the possibilities for teaching. We discuss our experiences with this approach. It appears that this language is not only very suited to illustrate the different approaches to make DSLs, but also an excellent motivation to introduce many advanced functional programming concepts.

Dear reviewer,

This paper is obviously not in the state we expected to reach before the deadline. Our apologies for that. We hope that current content and the outline provided by the empty sections will provide a sufficient clear idea of what we are planning to write. Since the contents is used for our course we are able to provide a complete version before the workshop. Of course we will accept a rejection based on this incomplete version.

---

1 Introduction

At the Radboud University Nijmegen there is an obligatory course functional programming, FP, in the bachelor computer science that covers the basis of functional programming using Clean [9]. In the master computer science there is a course advanced functional programming, AFP, that students follow depending on their choice for one of the specific tracks. The course AFP used to contain an actual collection of more advanced topics in functional programming like generic programming [1] and GADTs. It illustrates the use of functional programming concepts in application areas like task oriented programming in iTask [8] and model-based testing with Gast [4]. In order to improve the visibility of our master in computer science it has to become more coherent and to get a clearer profile. As part of the changing profile of the software track in the master education, the course AFP should focus more clearly on domain specific languages, DSLs, [2, 6]. This paper reports on the first changes made in the course AFP and the student experiences with these changes.

We start out by explaining that there are two different approaches for the realisation of a DSL. First, one can define such a language from scratch and make an implementation of the DSL by an interpreter or a compiler. For many applications this approach induces too much overhead. The additional work is caused by the design and implementation of boilerplate parts of the DSL like numerical expressions and control structures. The construction of such a language implementation is handled in the master course compiler construction. The second approach to construct DSLs avoids the design and implementation of

these boilerplate parts by embedding the the DSL in some existing host language. The DSL inherits he functionality of the host language and one can concentrate on the new features needed in the DSL. The new functionality of the DSL is obtained by constructing a library containing the types, operators and manipulation functions required in the DSL.

Next we present the students with two choices for the realisation of a DSL embedded in a functional programmings language. First, the DSL can be represented by its abstract syntax tree. Such a representation is similar to the representation of the DSL within a compiler for that language. Using the features of functional languages the effort to construct a tailor made tree is very limited. This approach is called a deep embedding of the DSL [3]. In Section 2 of this paper we elaborate on the deep embedding of DSLs. In our course we use the simple imperative language While as introduced by Neilson and Neilson [7] as our running example. The second approach to construct an embedded DSL embedded is by a set of functions and operators in the functional host language. This is called a shallow embedded DSL. In Section 3 we sketch the shallow embedding of While. For each of those approaches we list teaching opportunities. These topics can be part of the lectures as well as exercises or projects for the students.

In Section 4 we sketch some of the more complex DSLs used in this course and what we teach the students about those languages. In particular we introduce the iTask system for task oriented programming, and the logical as well as the state based DSLs used in model-based testing.

The experiences of the students with this approach are briefly discussed in section 5.

2 Deep Embedded DSLs

In a deep embedding the DSL is is represented by a tailor-made algebraic data structure of the host language. This concept is certainly not new. One of the famous examples is introduced in 1992 by Nielson and Nielson in their book *Semantics with applications* [7]. In their book they demonstrate that there is an almost one-to-one mapping between their formal secantics and an implementation in Miranda. Most of our students know this approach to semantics, since they use the vary same book in their bachelor course on semantics.

The simplest imperative language in this book is called While. Since such an imperative language does not fit directly into the functional framework ie is very suited as example DSL. It shows clearly how the DSL can be different from the host language and add its own features. This language is small enough to cover the complete implementation in various ways. On the other hand is is big enough to illustrate many interesting aspects of an embedded DSL. The simplest implementation of a DSL is an almost direct implementation of the semantics of the DSL.

2.1 The Running Example DSL While

The Nielsons show an implementation of the operational and denotational semantics of While in Miranda in the appendices of their book. Basically our deep embedding follows these implementations. We use Clean as an implementation language, and employ the features of a modern functional programming language whenever appropriate. Among the new features used are strong typing, user defined infix operators, overloading and generic programming.

2.2 Deep Embedding of While

The language While has arithmetic expressions a of type **Aexp**. We construct a type `Aexp` to mimic those expressions. In the representation in Clean we add some details about the representation of numbers and

variables, as well as the binding power of the infix operators to avoid parentheses in expressions as much as possible. We reproduce the definition of a from [7] on the left, and our representation on the right.

$$
\begin{array}{lcll}
a & ::= & n & // \text{ numeral } \textbf{Num} \\
 & | & x & // \text{ variable : } \textbf{Var} \\
 & | & a_1 + a_2 & \\
 & | & a_1 * a_2 & \\
 & | & a_1 - a_2 &
\end{array}
$$

```
:: Aexp = Num Num
        | Var Var
        | (+.) infixl 6 Aexp Aexp
        | (*.) infixl 7 Aexp Aexp
        | (-.) infixl 6 Aexp Aexp
:: Num  :== Int
:: Var  :== String
```

We have added a dot to the name of the infix operators in order to avoid a name class with the operators from the `StdEnv` of Clean. It is possible to omit the import of the basic operators from the standard environment and use these operators in the DSL. However, we discovered that such an approach causes easily confusion by the students and hence does more harm than good.

In the same way we define an algebraic type for the Boolean expressions in our DSL While. In `Bexp` we use constructors `TRUE` and `FALSE` instead of the basic type `Bool`, just to illustrate that it is not necessary to use basic types. Actually, it is more convenient to use a parameterized constructor `Bool Bool` than the constants `TRUE` and `FALSE`.

$$
\begin{array}{lcl}
b & ::= & \textbf{true} \\
 & | & \textbf{false} \\
 & | & a_1 = a_2 \\
 & | & a_1 \leq a_2 \\
 & | & \neg\, b \\
 & | & a_1 \wedge a_2
\end{array}
$$

```
:: Bexp = TRUE
        | FALSE
        | (=.) infix 4 Aexp Aexp
        | (<.) infix 4 Aexp Aexp
        | ¬. Bexp
        | (&&.) infixr 3 Bexp Bexp
```

In the same style we define an algebraic type for the statements of While. The differences between the original version and the representation in Clean are again slightly different names for operators and uppercase identifiers for the constructors representing the keywords.

$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \textbf{skip} \\
 & | & S_1 \,;\, S_2 \\
 & | & \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \\
 & | & \textbf{while } b \textbf{ do } S
\end{array}
$$

```
:: Stmt = (:=.) infix 2 Var Aexp
        | Skip
        | (:.) infixr 1 Stmt Stmt
        | IF Bexp THEN Stmt ELSE Stmt
        | While Bexp DO Stmt

:: THEN = THEN
:: ELSE = ELSE
:: DO   = DO
```

It is completely standard to define several function operating on these data structures. Useful functions are for instance a pretty printer, a function finding the variables occurring in some While-term, and a function that checks whether all variables are initialised before their first use.

Also the various versions of the semantics defined by the Nielsons are just manipulation functions of these data structures. In this paper we show only the big step operational semantics, but in our lecture we present the various versions of the semantics.

2.2.1 Operational Semantics

The big step operational semantics, also called natural semantics, is just an evaluator for the data types introduced above. In first approach the implementations follows the given semantics directly. Both versions use a state. This state is a function mapping variables to numbers. The semantics uses the a pattern match with the Scott-brackets: $\mathcal{A}[\![a]\!]$. In the realisation in Clean we use a function `A` that follows the semantics closely. We use an uppercase function name, `A`, in correspondence with the semantic function $\mathcal{A}$.

$$
\begin{aligned}
\mathbf{State} &= \mathbf{Var} \to \mathbf{Z} \\
\mathcal{A} : \mathbf{Aexp} &\to (\mathbf{State} \to \mathbf{Z}) \\
\mathcal{A}[\![n]\!]\ s &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]\ s &= s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]\ s &= \mathcal{A}[\![a_1]\!]\ s + \mathcal{A}[\![a_2]\!]\ s \\
\mathcal{A}[\![a_1 * a_2]\!]\ s &= \mathcal{A}[\![a_1]\!]\ s * \mathcal{A}[\![a_2]\!]\ s \\
\mathcal{A}[\![a_1 - a_2]\!]\ s &= \mathcal{A}[\![a_1]\!]\ s - \mathcal{A}[\![a_2]\!]\ s
\end{aligned}
$$

```
:: State := Var → Num

A :: Aexp State → Num
A (Num n)    s = n
A (Var x)    s = s x
A (a1 +. a2) s = A a1 s + A a2 s
A (a1 *. a2) s = A a1 s * A a2 s
A (a1 -. a2) s = A a1 s - A a2 s
```

For the Boolean expressions we use a very similar approach. By using the basic type for Booleans, `Bool`, and the associated operators as a result, the formulation in Clean is more concise than the original semantics.

$$
\begin{aligned}
\mathcal{B} : \mathbf{Bexp} &\to (\mathbf{State} \to \mathbf{T}) \\
\mathcal{B}[\![\mathbf{true}]\!]\ s &= \mathbf{tt} \\
\mathcal{B}[\![\mathbf{false}]\!]\ s &= \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]\ s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]\ s = \mathcal{A}[\![a_2]\!]\ s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]\ s \neq \mathcal{A}[\![a_2]\!]\ s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]\ s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]\ s \leq \mathcal{A}[\![a_2]\!]\ s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]\ s > \mathcal{A}[\![a_2]\!]\ s \end{cases} \\
\mathcal{B}[\![\neg b]\!]\ s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]\ s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]\ s = \mathbf{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]\ s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]\ s = \mathbf{tt} \\ & \text{and } \mathcal{B}[\![b_2]\!]\ s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]\ s = \mathbf{ff} \\ & \text{or } \mathcal{B}[\![b_2]\!]\ s = \mathbf{ff} \end{cases}
\end{aligned}
$$

```
B :: Bexp State → Bool
B TRUE         s = True
B FALSE        s = False
B (a1 =. a2)   s = A a1 s == A a2 s
B (a1 <. a2)   s = A a1 s ≤ A a2 s
B (¬. b)       s = not (B b s)
B (b1 &&. b2)  s = B b1 s && B b2 s
```

For the operational semantics of While we need to be able to update the state. A direct, but inefficient, definition that is very close to the semantical functions is the infix operator `|->` which is printed as $\mapsto$:

$$
(s\ [y \mapsto v])\ x = \begin{cases} v & \text{if } x = y \\ s\ x & \text{if } x \neq y \end{cases}
$$

```
(↦) infix :: Var Num → State → State
(↦) y v = λs x. if (x == y) v (s x)
```

The semantic function for the natural semantics changes the state. It is a partial function since it is not defined for nonterminating statements in While (like `While TRUE DO Skip`). In Clean we cannot reason about nontermination in the same way, here the undefinedness is indicated by a nonterminating computation.

$$\mathcal{S}_{ns} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$
$$\mathcal{S}_{ns}[\![S]\!]\ s = \begin{cases} s' & \text{if } \langle S,\ s\rangle \to s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

```
ns :: Stmt State → State
ns (v :=. a)  s = (v ↦ A a s) s
ns (s1 :. s2) s = ns s2 (ns s1 s)
ns Skip       s = s
ns (IF c THEN t ELSE e) s =
    if (B c s) (ns t s) (ns e s)
ns stmt=:(While c DO body) s =
    if (B c s) (ns stmt (ns body s)) s
```

The relation $\langle S,\ s\rangle \to s'$ is given by the famous horizontal bars from semantical descriptions. The rules for **While** are the most tricky examples due to the potential undefinedness:

$$\frac{\langle S,\ s\rangle \to s',\ \langle \textbf{while } b \textbf{ do } S,\ s'\rangle \to s''}{\langle \textbf{while } b \textbf{ do } S,\ s\rangle \to s''} \text{ if } \mathcal{B}[\![b]\!]\ s = \textbf{tt}$$
$$\langle \textbf{while } b \textbf{ do } S,\ s\rangle \to s \text{ if } \mathcal{B}[\![b]\!]\ s = \textbf{ff}$$

The denotational and small-step semantics are defined in a very similar way.

2.3 Opportunities for Teaching

Based on this complete implementation of the simple language While one can teach many interesting topics. The differences between the detection undefinedness and nontermination is in the semantics functions $\mathcal{S}_{ns}$ and `ns` in touched above. By implementing the other versions of the semantical functions it becomes easy to explain their differences and similarities as well as to experiment with those functions.

Other Views of the Language

For DSLs specified by a data structure it is straightforward to define other functions over this data structure. A function used in the book of Nielson And Nielson collects the free variables in expressions. In a similar way we collect the variables used in assignments since they are particular useful to show show their value in the state produced by the semantics.

```
vars :: Stmt → [Var]
vars (x :=. e)           = [x]
vars (s :. t)            = vars s + vars t
vars Skip                = []
vars (IF c THEN t ELSE e) = vars t + vars e
vars (While c DO b)      = vars b
```

```
instance + [x] | Eq, Ord x where (+) x y = sort (removeDup (x ++ y))
```

Examples of other useful functions are pretty printers, equality, and transformation of expressions and statements in While.

Generic Functions for While

Generic programming is another important topic in our advanced functional programming course. It is obvious that manipulations like pretty printing and equality for `Aexp`, `Bexp` and `Stmt` can be derived. In order to test properties about the semantics that quantify over the elements from While we also need to

generate these elements. To prevent undesirable language constructs, like nonterminating statements, we use a tailor made generation using an additional data type [5].

Testing Properties about While with Gast

Since the semantics is now an executable specification, we can use it to evaluate programs in While. We can easily turn this into properties tested by Gast. For instance the Euclidean algorithm to compute the greatest common divisor is given by `gcdStmt`. The property `propGCD` states that its result should be equal to the result computed by the function `gcd` from the standard environment of Clean. The `Start` rule tests this property for small integer values between 0 and 42. We use a tailor made test suite generation in order to avoid very long computations.

```
gcdStmt :: Stmt
gcdStmt =
  IF (Var "a" =. Num 0)
    THEN ("c" :=. Var "b")
    ELSE
      (While (¬. (Var "b" =. Num 0)) DO
        (IF (Var "a" <. Var "b")
          THEN ("b" :=. Var "b" -. Var "a")
          ELSE ("a" :=. Var "a" -. Var "b")
        ) :.
         "c" :=. Var "a"
      )

propGCD :: (Int, Int) → Bool
propGCD (a, b) = and (ns gcdStmt (("a" ↦ a) (("b" ↦ b) emptyState))) "c" == gcd a b

Start = test (propGCD For [(n, m) \\ n ← [0..max], m ← [0..max]]) where max = 42
```

As stated above we have to be careful to avoid nonterminating statements in properties that quantify over statements in While. In the course we show how one can define statements in While that will terminate by using and additional data structure that represents only loops with a decreasing counter. This is an application of the technique outlined in [5].

Hiding the State in a Monad

In the definitions above the state is a function from variables to values that is moved around explicitly. Nowadays it is more custom to hide the state in a monad and pass it around more implicitly. This can be demonstrated by defining the usual bind and rerun functions for the a state monad. The additional functions `read` and `write` access the state.

```
:: Sem a := State → (a, State)     // Semantics

(>>=) infix 0 :: (Sem a) (a→Sem b) → Sem b
(>>=) f g = λs.let (a, x) = f s in g a x

rtn :: a → Sem a
rtn a = λs.(a, s)

read :: Var → Sem Int
```

```
read v = λs.(s v, s)

write :: Var Int → Sem Int
write v x = λs.(x, (v ↦ x) s)
```

The semantics of expressions and statements can be adapted to this monad. Here we use a class `sen` for the semantics instead of three different functions. We price to be paid for a such a class is that we have to provide some additional information to the type checker in order to solve the overloading. This is the reason we have to write `y+0` instead of just `y` in the alternatives for the =. and <. operators.

```
class sem a b :: a → Sem b

instance sem Aexp Int where
    sem (Num n)    = rtn n
    sem (Var x)    = read x
    sem (a1 +. a2) = sem a1 >>= λx.sem a2 >>= λy.rtn (x + y)
    sem (a1 *. a2) = sem a1 >>= λx.sem a2 >>= λy.rtn (x * y)
    sem (a1 -. a2) = sem a1 >>= λx.sem a2 >>= λy.rtn (x - y)

instance sem Bexp Bool where
    sem (Bool b)    = rtn b
    sem (a1 =. a2)  = sem a1 >>= λx.sem a2 >>= λy.rtn (x == y+0)
    sem (a1 <. a2)  = sem a1 >>= λx.sem a2 >>= λy.rtn (x ≤ y+0)
    sem (¬. b)      = sem b  >>= λx.rtn (not x)
    sem (b1 &&. b2) = sem b1 >>= λx.sem b2 >>= λy.rtn (x && y)
```

A more Efficient State

The state introduced above works fine for small examples, but its size and grows linear with the number of assignments. Using an explicit access function `read` instead of function application enables us to use a more efficient implementation like a binary search tree.

Variable of various Types

Limitations of Type System of the Host Language for more general Expressions

.

How to Add Functions to the DSL and the Associated Name Spaces

.

The Consequences of Lazy Evaluation

.

Simulation with iTask

3 Shallow Embedded DSLs

.

3.1 Deep Embedding of While

.

3.2 Opportunities for Teaching

.

Comparing the Advantages and Disadvantages of the Shallow and Deep Embedding

.

State Manipulation is Independent of the Representation of the DSL

.

The Need of GADTs for some Complex Manipulations

.

Requiring multiple Views Imposes Limitations

.

The Consequences of Language Extensions

.

The Necessity of a State for Assignments

.

Shallow and Deep Embedding can be mixed in a single DSL

4 More Complex DSLs

.

4.1 The iTask System

.

4.2 Model-Based Testing

.

4.3 Opportunities for Teaching

.

Why many Practical DSLs are Shallow Embedded

.

Why many Practical DSLs have only a single View

.

Exercises in the Domain of the DSLs used as Examples

.

The Type Tells Everything about Constructs in the DSL

5 Student Experiences

We had 28 students in this edition of our course. Most of the students appreciate the DSL centred approach to the course in advanced functional programming. In their feedback the explicitly indicate that they learn about DSLs as well as advanced functional programming concepts in a useful context.

However, two students did not like the course at all. The claim to have seen everything about semantics and the language While in de bachelor course on semantics. The more advanced DSLs like iTask and Gast are classified as our academic toys that serve no useful purpose at all. To cope with this criticism we will try outline the role of While as well as the advanced DSLs more clearly in the next iteration of this course.

6 Discussion

In this paper we discuss how our advanced functional programming course is centred around domain specific languages and the way to embed them in a functional host language. Apart from the useful knowledge about domain specific languages it also provides an excellent motivation to introduce and apply many advanced functional language concepts.

References

[1] Artem Alimarine & Rinus Plasmijer (2001): *A generic programming extension for Clean*. In Thomas Arts & Markus Mohnen, editors: *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL '01, Stockholm, Sweden*, Ericsson Computer Science Laboratory, pp. 257–278.

[2] Martin Fowler (2010): *Domain Specific Languages*, 1st edition. Addison-Wesley Professional.

[3] Jeremy Gibbons (2013): *Functional Programming for Domain-Specific Languages*. In Viktoria Zsok, editor: *Central European Functional Programming - Summer School on Domain-Specific Languages*. Available at `http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/fp4dsls.pdf`.

[4] Pieter Koopman, Artem Alimarine, Jan Tretmans & Rinus Plasmeijer (2003): *Gast: generic automated software testing*. In Ricardo Peña & Thomas Arts, editors: *Revised Selected Papers of the 14th International Workshop on the Implementation of Functional Languages, IFL '02*, *LNCS* 2670, Springer-Verlag, pp. 84–100.

[5] Pieter Koopman & Rinus Plasmeijer (2006): *Automatic Testing of Higher Order Functions*. In Naoki Kobayashi, editor: *Proceedings of the 4th Asian Symposium on Programming Languages and Systems, APLAS '06*, *LNCS* 4279, Springer-Verlag, Sydney, Australia, pp. 148–164.

[6] Marjan Mernik, Jan Heering & Anthony M. Sloane (2005): *When and How to Develop Domain-specific Languages*. *ACM Comput. Surv.* 37(4), pp. 316–344, doi:10.1145/1118890.1118892. Available at `http://doi.acm.org/10.1145/1118890.1118892`.

[7] Hanne Riis Nielson & Flemming Nielson (1992): *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc.

[8] Rinus Plasmeijer, Peter Achten & Pieter Koopman (2007): *iTasks: executable specifications of interactive work flow systems for the web*. In Ralf Hinze & Norman Ramsey, editors: *Proceedings of the International Conference on Functional Programming, ICFP '07*, ACM Press, Freiburg, Germany, pp. 141–152.

[9] Rinus Plasmeijer & Marko van Eekelen (2001): *Concurrent Clean language report (version 2.0)*. `http://www.cs.ru.nl/~clean/`.