The design and implementation of embedded Domain-Specific Languages

Pieter Koopman & Mart Lubbers

{pieter,mart}@cs.ru.nl

Radboud University

IN·DEI·NOMINE·FELICITER

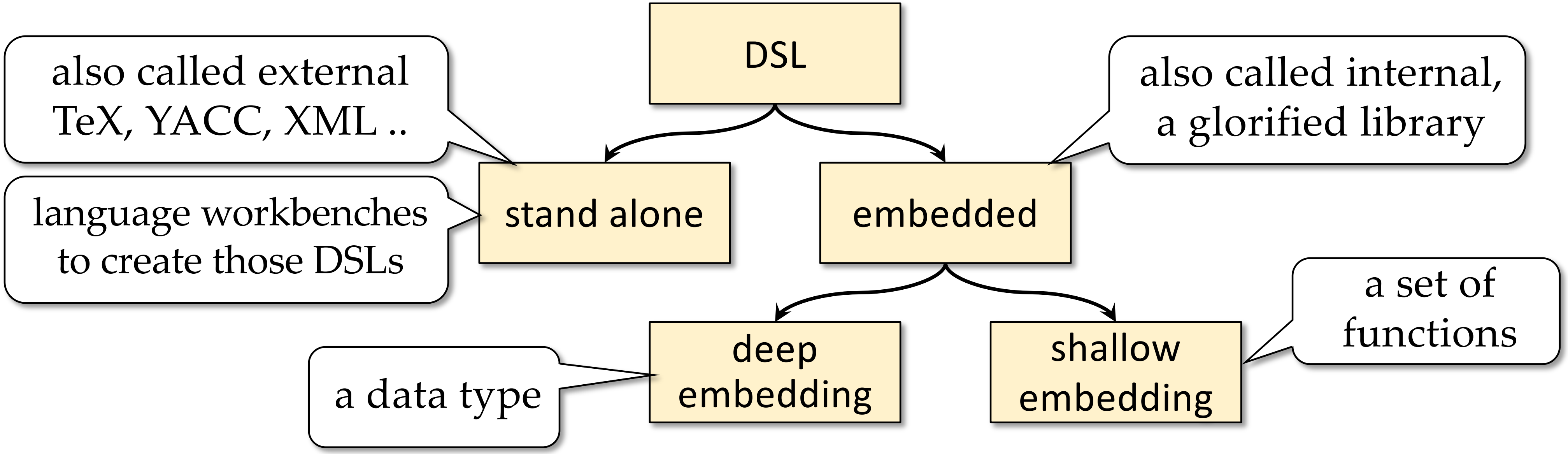
Domain-Specific Languages, DSLs

- DSL is specialized programming language for a particular application domain
  - e.g. **HTML** for webpages, **TeX** for type setting, **SQL** for database queries, **SVG** for graphics, GraphViz's **dot** for visualisation of graphs, **YACC** parser construction, **QuickCheck** for properties in model-based testing, …

- in contrast to a general-purpose language (GPL)
  - e.g. Haskell, Java, Erlang, Scala, C/C++, …

- Why a DSL?
  - **make programming and maintenance easier and cheaper**
  - DSL offers abstraction level tailored to problem domain
  - DSL implementation avoids repeated work

implementation strategies for DSLs



- we focus on embedded DSLs
  to reuse host language implementation, libraries and tools
- Two concepts: **views**, DSL **constructs**

implementation goals for eDSL

1. Multiple views
   - execution (simulation), and pretty printing, code generation, …
   - adding views should not break anything in existing views and DSL programs
2. Type safety
   - eDSL has same level of trust as host language
   - no runtime type errors
   - eDSL constructs can be overloaded (like equality, comparison, addition)
3. Safe identifiers
   - No runtime errors (so no strings or numbers as identifiers)
4. Extendable DSL
   - extend DSL without breaking existing code (programs and views)

excludes shallow embedding (functions)

Wadler's expression problem

eDSL and running example

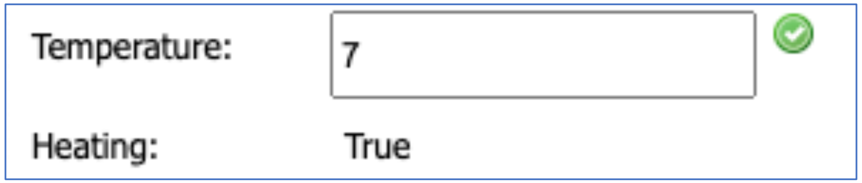
- eDSL to control peripherals (temperature sensor, heating device, …)
- eDSL is functional and task-oriented

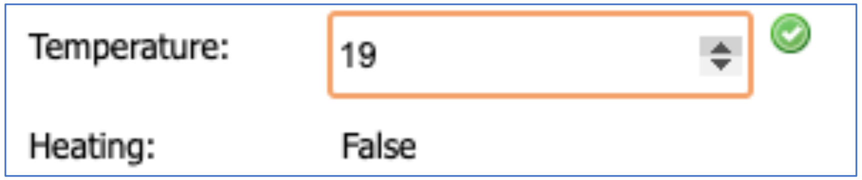
```
program1 =
    define sens
    define heat
    Loop (
        Read sens >>=. \temp ->
        Post (temp < 19) heat
    )
```

how to define these identifiers?

host language
our eDSL

- simulation view in the browser of this program





1. multiple views
2. type safety
3. safe identifiers
4. extendable DSL

Deep embedding

our example eDSL in a deep embedding

- Deep embedding
  - language constructs as constructors *of* an ADT
  - views (interpretations) as functions *over* the ADT
  - hosted in Clean, a language similar to Haskell.

Nicholas Rinaudo's talk of day 1

- views of the eDSL we will make
  - pretty printing (code generation is essentially equal)
  - simulation in web-browser (using the iTask framework)

Plasmeijer's tutorial of day 1

Expression DSL

```
:: Expr a
   = Lit a
   | E.b: Less (Expr b) (Expr b) & <, type b -> Expr Bool
   ...
```

argument fixes type of expression

class restrictions on type variable

expression yields a Boolean: GADT

// add any operation you need in the DSL

- we need quite some machinery to make this safe and looking good
  - GADT, existential quantified type variables, class restrictions in types, infix constructors
  - explicit projection pairs/bimaps can replace GADTs
  - languages like Haskell and Clean are made for this

Simple views on the expression DSL

```
print :: (Expr a) -> String | type a
print (Lit a)     = toString a
print (Less l r) = print l +++ "<" +++ print r
print ...         = ...

eval :: (Expr a) -> a
eval (Lit a)     = a
eval (Less l r) = eval l < eval r
eval ...         = ...
```

using GADTs

1. multiple views
2. type safety
3. safe identifiers
4. extendable DSL

Higher-order abstract syntax, HOAS

Higher-order abstract syntax (HOAS)

- Use functions in the host language to define variables
  - Nameless functions (lambdas) are useful but not required
- use higher-order programming
  - What would you do with a value if you had it as some point (monads without do)

    ```
    do { x <- compute; f x; }
    ```

    this is a shorthand notation for

    ```
    compute >>= \x -> f x
    ```

- We can still print, evaluate, simulate, …
  but transformations are more tricky

expressions in our task-oriented eDSL

```
w2Deep :: Work Bool
w2Deep =
    DefSens "Temperature" 7 \sens ->
    DefActr "Heating" False \heat ->
    Loop (
        Read sens >>=. \temp ->
        Post ( Less temp (Lit 19) ) heat
    )
```

host language

our eDSL

our eDSL in a deep embedding

```
:: Sens a = ...
:: Actr a = ...
:: Work a
    = E.b: DefSens String b ((Sens b) -> Work a) & type b
    | E.b: DefActr String b ((Actr b) -> Work a) & type b
    | Read (Sens a)
    | Post  (Expr a) (Actr a)
    | E.b: (>>=.) infixl 1 (Work b) ((Expr b)->Work a) & type b
    | Loop (Work a)
    | (.||.) infixr 3 (Work a) (Work a)
```

quantified type variable

HOAS for safe identifiers

class restriction

infix constructor

sequencing work

repeating work

parallel work

Deep embedding: simulation

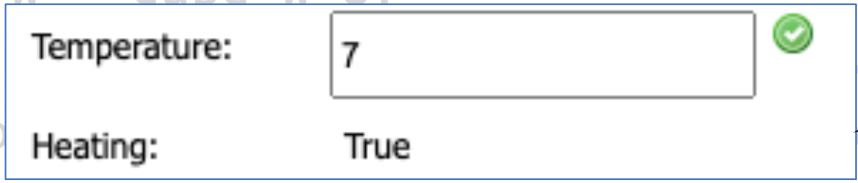
```
host language
our eDSL
the iTask eDSL
```

```
sim :: (Work a) -> Task a | type a // big step
sim w = case w of
  w >>=. f = sim w >>- \a -> sim (f (Lit a ))
  Loop w   = sim (w >>=. \_ -> Loop w)
  v .||. w = sim v -||- sim w
  DefSens n v f =
    withShared v \sds ->
    (Label n @>> updateSharedInformation [] sds) ||-
    sim (f (Sens (Sds sds)))
  Read (Sens (Sds s)) = get s
  ...
```

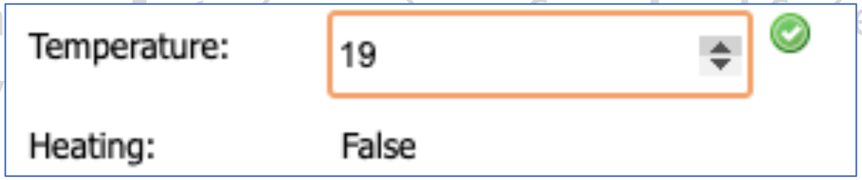
itasks
task-oriented programming framework

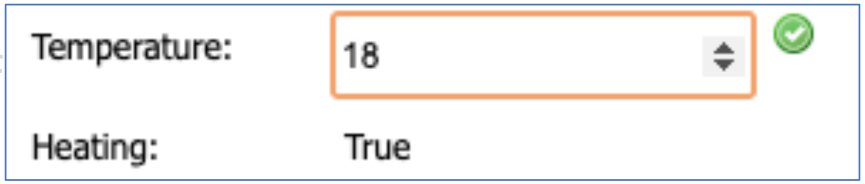
Deep embedding: simulation

```
sim :: (Work a) -> Task a | type a // big step
sim w = case w of
  w                                ay                     a ))
  Lo                                oop w)
  v Till f =
    sim (v >>=. \a                                 eval c) u w)
  v .||. w = sim v
  DefSens n v f =
    withShared v \sds ->
    (Label n @>> updateSharedInf
    sim (f (Sens (Sds sds)))
  Read (Sens (Sds s)) = get s
  ...
```



running in browser





review of deep embedding

1) multiple views
   - just add a function over our datatypes
   - clear distinction between host language and eDSL
2) type safety
   - type-class system of host language takes care of this
   - we use GADTs, extensional quantified type-variables, ..
   - intensional analysis is easy
3) safe identifiers
   - use higher-order functions (HOAS)
4) extendable DSL 😰
   - change the datatypes, limited compiler support to adjust all views

1. multiple views
2. type safety
3. safe identifiers
4. extendable DSL

class-based embedding, a set of type-constructor classes

tagless-final embedding

expressions

```
class expr v where
    lit :: a -> v a | type a
    (<.) infix 4 :: (v b) (v b) -> v Bool | <, type b

:: Print a = P String       // print view
instance expr Print where
    lit a = P (toString a)
    (<.) a b = P (uP a +++ " <. " +++ uP b)

:: Eval a = Eval a          // simulation view
instance expr Eval where
    lit a = pure a
    (<.) a b = (<) <$> a <*> b
```

types are a bit simpler than in the deep embedding

actual implementation has source of fresh identifiers

the Eval monad makes this more concise

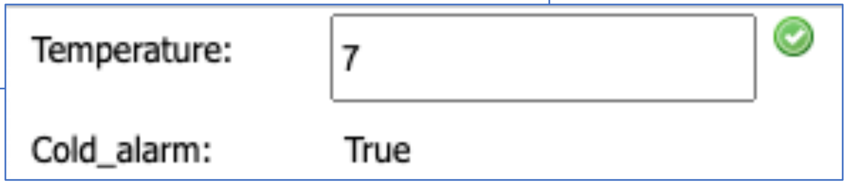
our running example

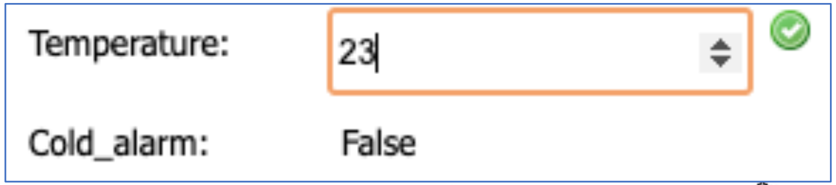
```
:: WorkC v a :== v (Task a)

w2Class :: WorkC v Bool | dsl v
w2Class =
    defSensC "Temperature" 7    \sens ->
    defActrC "Cold_alarm" False \alarm ->
    loopC (
        readC sens >>=.. \temp ->
        postC (temp <. lit 15) alarm
    )
```

functions instead of datatypes

```
defSensC Temperature 7 ->
defActrC Cold_alarm False ->
loopC (
  Read Temperature >>=.. \v0 ->
  Post (v0 <. 15 ) Cold_alarm
)
```





looping and defining sensors + printing

```
class loopC v :: (WorkC v a) -> WorkC v a | type a
class sens v where
  readC :: (SensC v a) -> WorkC v a | type a
  defSensC :: String b ((SensC v b) -> WorkC v a) -> WorkC v a| …

instance loopC Print where
  loopC w = P ("loopC " +++ uP w)

instance sens Print where
  readC (SensC n) = P ("Read " +++ uP n)
  defSensC name val f
   = P (print ["defSensC ",name," ",toString val," ->"] +++
     uP (f (SensC (P name))))
```

looping and defining sensors + simulation

```
class loopC v :: (WorkC v a) -> WorkC v a | type a
class sens v where
  readC :: (SensC v a) -> WorkC v a | type a
  defSensC :: String b ((SensC v b) -> WorkC v a) -> WorkC v a| …

instance loopC Eval where loopC w = w >>=.. \_ -> loopC w

instance sens Eval where
  readC (SensC n) = n >>= pure o get
  defSensC name val f
  = pure (withShared val \sds -> (Label name @>>
    updateSharedInformation [] sds) ||- uE (f (SensC (pure sds))))
```

itasks
task-oriented programming framework

review of class-based embedding

1) multiple views
   - adding a view is just making a new class instance

2) type safety
   - type-class system of host language takes care of this
   - types are easier than for deep embedding (datatypes)
   - intensional analysis is more tricky

3) safe identifiers
   - reuse our technique with higher-order functions

4) extendable DSL
   - just add a class to extend the eDSL
   - good compiler support to check that required instances exist

overview

	shallow embedding	deep embedding	class-based embedding
	functions	datatypes	classes
evaluation	✅	✅	✅
printing/code	❌	✅	✅
multiple views	❌	✅	✅
eDSL optimization	❌	✅	😰 rather tricky
simple types	✅	🤔 GADTs etc.	✅
good type errors	✅	✅	🤯 in terms of classes
extend eDSL	✅	❌ 1)	✅
easy to add views	❌	✅	✅

1) Deep Embedding with Class doi: 10.1007/978-3-031-21314-4_3

what should be part of our DSL

eDSL design

adding constructs to eDSL

```
Loop w = w >>=. \_ -> Loop w

w Till f = ...
Read sens Till \temp ->
    (Less temp (Lit 19),Post (Lit True) heat)

w >>|. f = w >>=. \_ -> f


DefTemp f = DefSens "Temperature" 7 f
```

- YES
  - useful addition
  - other views less simple
- YES
  - useful addition
  - shorter and clearer
- yes ?
  - users might expect it
  - limited added power
  - as macro?
- no
  - limited added value
  - user can define it

adding constructs to eDSL

```
Loop w = w >>=. \_ -> Loop w

w Till f
Read sens
    (Less

w >>|. f =

DefTemp f =       "Temperature" 7 f
```

conditions:

- useful addition to the abstraction to offer
  - adds new power to the language
  - make eDSL programs clearer
  - make eDSL programs shorter
- things people might expect
- use host language as macro language

- YES
  - useful addition
  - other views less simple
- YES
  - useful addition
  - shorter and clearer
- es ?
  - users might expect it
  - limited added power
  - as macro?
- no
  - limited added value
  - user can define it

Domain Specific Languages - conclusions

- DSLs help you to create and maintain code in specific domain
  - tailor-made abstraction layer, separation of concerns
- embedded (internal) DSLs integrate with host language and reuse tooling
  - external DSLs require parser, type checker, code generator, libraries, tooling, …
- DSL implementation strategies – with their own advantages and challenges
  - shallow embedding   DSL is set of functions
  - deep embedding   DSL is datastructure (GADTs, HOAS, ..)
  - class-based embedding   DSL is set of type constructor classes
- DSL design
  - do not hesitate to add constructs that improve the creation or maintenance of code
  - keep the DSL as small as possible
  - use the host-language as macro mechanism
- there is much more on eDSLs than we can tell in this tutorial