Dynamic Creation of Well-Typed DSL Expressions

Pieter Koopman[1], Steffen Michels[2], and Rinus Plasmeijer[1,2]
pieter@cs.ru.nl, steffen@top-software.nl, rinus@cs.ru.nl

[1] Radboud University, Nijmegen, The Netherlands www.ru.nl/icis
[2] TOP Software Solutions, The Netherlands www.top-software.nl

Extended Abstract – Research Paper

Abstract. For interactive systems it is often desirable that users can create tasks for the system dynamically. Often these tasks are internally specified by constrained types like Generalized DataTypes, GADTs, or function applications using typeclasses. For plain datatypes, or the corresponding functions, this is relative easy: the input can be captured by a structured editor or a simple parser from a textual input. However, in many situations such simple types are not enough. We either need GADTs or more constraints than can be checked by a parser.
To guarantee correct inputs we either need the invoke the compiler of the host language and add the compiled input dynamically to the program, or we need implement a rather complicated type-checker for the input. Both solutions are complicated and require a significant of work. Fortunately, Clean provides an advanced type-system for its dynamics. The existing type-system for these dynamic values can check all required type constraints. In this paper we show how we can make dynamic editors for complex user inputs in iTask programs using these dynamic types.

Keywords: Dynamics · Web-editors · DSL.

1 Introduction

In the Task Oriented Programming, TOP, system iTask we can derive structured editors for ordinary datatypes by generic programming [**?**]. As usual in generic programming these data types should neither contain functions nor existentially quantified variables. This is wonderful to create programs in a simple deep embedded Domain Specific Language, DSL, dynamically. We define a datatype representing the DSL, derive a structured web-based editor for this datatype and we have the type-safe dynamic editor for our DSL.

It is well-known that ordinary datatypes are not powerful enough to capture all constraints for more complex DSLs. This was one of the main reasons to develop more complex datatypes like GADTs and introduce additional language features like quantified type variables in the datatypes and the associated class constraints. These extensions of the datatypes cannot be handled by the generic system for good reasons: such a datatype cannot be represented in the usual generic way. Nevertheless, such datatypes (or even better the corresponding functions) are required when we have DSL containing features like overloading without dynamic type-problems.

Typical systems requiring such a complex input are iTask systems where the user can create tasks for the problem domain dynamically. Another example is the mTask system that extends the iTask system with tasks that can be executed on tiny Internet of Things, IoT, devices [?]. These IoT devices are typically too small and too slow to execute full blown iTask programs. Nevertheless, we are able to execute heavily restricted tasks on such IoT devices and even to ship those tasks dynamically to the IoT devices [?]. The mTask system is a tagless DSL: the system consists of a set of type classes and instances for each interpretation of mTask programs [?]. It is often desirable to compose such mTask programs dynamically. Since the mTask DSL cannot be represented in a type safe way by a plain algebraic datatype, this is not possible without risking runtime type-errors.

In this paper we show how we can make typesafe DSL programs using a special variant of iTask editors. The basic idea is to provide a selection box in the web-editor where the user can select one of the elements of the DSL. The arguments of this DSL-construct are added later. The iTask editor produces a `Dynamic` function corresponding to the DSL construct that still requires the appropriate arguments. Next we use iTask editors to add the arguments of the desired type. Using the type of the arguments required by the generated `Dynamic` can select the basic type of the elements that should be enabled in the iTask editor. Type errors in the arguments are indicated in the web-interface as soon as the arguments are provided in the iTask editor. This approach requires more programmer effort that just deriving a web-editors for a plain datatype, but it solves the problem of dynamically generating DSL expressions with the required type constraints.

In Section 2 we briefly review the web-editors for plain ADTs in the iTask system. Section 3 shows how we can make a variant of GADTs suited to make type-safe DSLs and how to make a type-safe web-editor for such a type. In Section 4 we show how we make a web-editor for a simple version of iTask expressions in an iTask editor. Section 5 reveals some of the internals of the dynamic editors. Finally, we discus the results of this paper in Section 6.

2 Basic iTask web-editors

Web-based editors for arbitrary algebraic datatypes are one of the basic components of the iTask system. Such an editor is a basic task that enables the user to construct an instance of such a datatype. Using a combinator other tasks can decide to use the current value of the web-editor as the final result. Examples of such combinators are a `continue`-button to be pressed by the user and a step combinator with a continuation that steps based on the actual value in the web-editor.

To illustrate the use of web-editors we introduce a datatype to represent a small DSL with integer and Boolean values and a tiny set of operations.

```
:: EExpr
   = Int Int
   | Bool Bool
   | EVar String
   | EAdd EExpr EExpr        // integer addition
   | EAnd EExpr EExpr        // Boolean And
```

```
  | EEq  EExpr EExpr        // overloaded equality
  | EIf  EExpr EExpr EExpr  // conditional
```

In the iTask system we can create a web-editor for values of type `EExpr` by:

```
derive class iTask EExpr // generate all needed generic manipulations of the type

editEExpr :: Task EExpr
editEExpr = Title "Make an EExpr" @>> enterInformation []

Start world = doTasks editEExpr world
```

Some screenshots of resulting editors in an arbitrary browser look like:

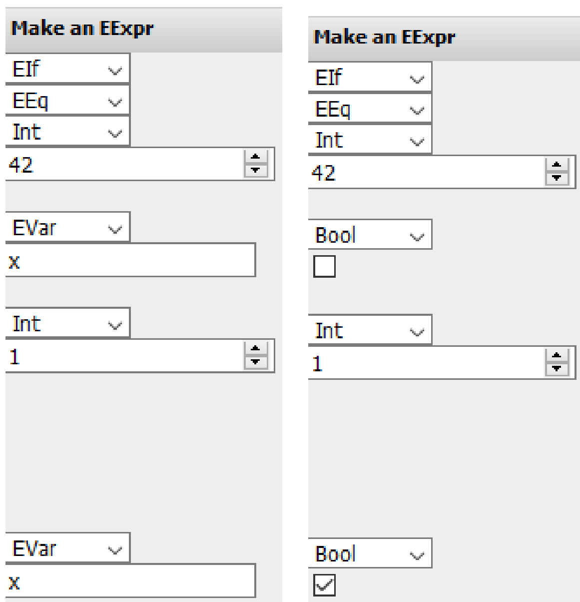

Fig. 1. Some screenshots of the editor for expression in use.

Obviously, the datatype `EExpr` and hence the web-editor for this type allows many expressions that should be rejected in strongly typed DSL. For instance, the expression `EAdd (Int 5) (Bool True)` is happily accepted.

3 Generalized Algebraic DataTypes

To enable the type system of the host language, here Clean, to check the types in our DSL we need a more advanced type to represent our DSL. Various versions of GADTs are proposed to solve this problem. Here we use a representation that does not require an extension of the type system.

```
:: Expr a
   = Lit a                                      // overloaded literal
   | Var (BM a Int) String                      // integer variable
   | Add (BM a Int) (Expr Int) (Expr Int)       // integer addition
   | And (BM a Bool) (Expr Bool) (Expr Bool)    // Boolean conjunction
   | ∃b: Eq (BM a Bool) (Expr b) (Expr b) & == b // overloaded equality

:: BM a b = {ab::a→b, ba::b→a} // BiMap to express the equality of the types a and b

bm :: BM a a                    // the only instance of BM that is ever used
bm = {ab = id, ba = id}
```

The erroneous expression from above in this representation is `Add bm (Lit 5) (Lit True)`. The Clean type-checker rejects this expression with the message `Type error in argument 3 of Add: cannot unify Expr Int with Expr Bool`. This is exactly the effect we want for our DSL. Moreover, the error message indicates the type problem rather well.

A typesafe evaluator for this DSL reads:

```
eval :: (Expr a) → a
eval expr = case expr of
    Int  bm i   = bm.ba i
    Bool bm b   = bm.ba b
    Var  bm s   = bm.ba 0
    Add  bm x y = bm.ba (eval x + eval y)
    And  bm x y = bm.ba (eval x && eval y)
    Eq   bm x y = bm.ba (eval x == eval y)
```

Problems start when we want to use a web-editor for the type `Expr a`. There are actually various problems:

1. The system cannot decide the type of the type argument `a` and hence cannot make an editor for it. This is very similar to making an editor for type `[a]`, the iTask system cannot make an editor for it since it is unclear what editor must be used for `a`. We can cope with this problem by choosing an specific argument like make on editor for `[Int]` or `Expr Bool`.
2. The type `Expr` uses the type `Int`, `Bool` and `BM a b`. In order to derive the class `iTask` for `Expr` we need instances for all those types. For the basic types `Int` and `Bool` this is not a problem, the iTask library provides appropriate instances. Our type `BM` contains functions and hence it is impossible to derive a web-editor for it. Since we only need the instance `bm` for this type, we can try to define some appropriate instance by hand.
3. The hardest problem is that we need an existentially quantified type variable `b` to express the overloading of the equality, `Eq`, in our DSL. The iTask system has no clue what type is required here and hence cannot make editors for the arguments of the conditional.
   Note that this is not a problem for the addition, `Add`, and logical conjunction, `And`. The type `Expr a` indicates the required types of the arguments here; `Int` and `Bool` respectively.

In order to make an editor for this type we use a lower abstraction level of the iTask system. We manually code a drop-down menu containing relevant typed constructs in the DSL. When the user selects such a construct the system generate a dynamic function that will produce the corresponding DSL construct when it has received the correct type arguments. The system uses the dynamic types to select the applicable elements in the drop-down menu.

For instance the basic case and the relevant integer expressions are made by:

```
exprEditor :: DynamicEditor (Expr a) | type a
exprEditor = DynamicEditor
   [ // This cons is used to provide untyped ‘TaskExpr‘ values.
     DynamicCons $ functionConsDyn "Expr" "(enter expr)"
        (dynamic λ (Typed expr) → expr :: (Typed (Expr a^) a^) → (Expr a^))
        <<@ HideIfOnlyChoice
   , DynamicConsGroup "Int"
        [ functionConsDyn "Int" "an integer value"
            (dynamic λi → Typed (Int bm i) :: Int → Typed (Expr Int) Int)
            <<@ HideIfOnlyChoice
        , functionConsDyn "Add" "add"
            (dynamic λ (Typed x) (Typed y) → Typed (Add bm x y) ::
            (Typed (Expr Int) Int) (Typed (Expr Int) Int) → Typed (Expr Int) Int)
            <<@ applyHorizontalBoxedLayout <<@ HideIfOnlyChoice
        , functionConsDyn "Var" "variable"
            (dynamic λ (Typed s) → Typed (Var bm s) ::
            (Typed String String) → (Typed (Expr Int) Int)) <<@ HideIfOnlyChoice
        ]
   ]
```

Some screenshots of the dynamic editor depicts the generated user interface are depicted in Figure 2.

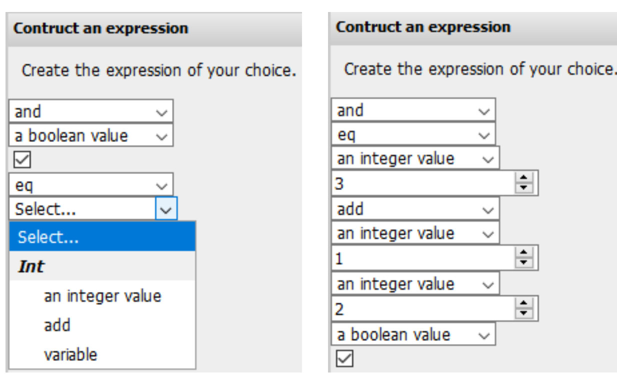

Fig. 2. Screenshots of the editor for typesafe expressions.

4 Function Editors

For functions, like iTask or mTask expressions, instead of datatypes we have a very similar problem as for our DSL type `Expr a`. It is impossible to make generic editors for functions and hence editors for iTask and mTask expressions. Fortunately, we can use a very similar solutions as in the previous section: we define a list of editors by hand to produce relevant dynamics representing the relevant functions. The very same dynamic machinery is used to select relevant editors and to check the type constraints dynamically.

An example works, but has to be added to this paper. Some screenshots to illustrate the possibilities to enter a simple task and to execute it are given in Figure 3.

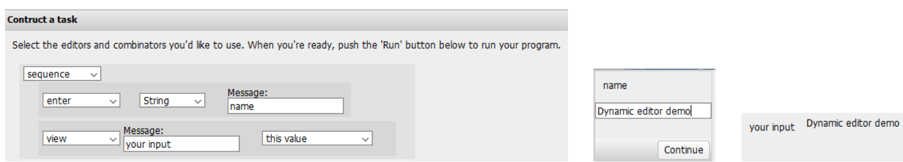

Fig. 3. Some screenshots of an editor for iTasks and the execution of the created iTask.

5 Dynamic Editors

In the final version of this paper this section reveals the internal details of the dynamic editor extension of the iTask system.

6 Discussion

To represent strongly typed deep or shallow embedded DSL plain algebraic datatypes are insufficient. We need some form of generalized ADTs or function applications guarantee type safe DSL expression by type system of the host language. It is often convenient to create DSL programs dynamically instead of as a part of a program in the host language. Unfortunately, the web-editors are not able to handle GADTs and function composition. In this paper we introduced a powerful workaround. We showed how we can make handcrafted editors that produce dynamic values and how the native type support for these dynamics can be used to check the types in our DSL dynamically. This allows us to compose typesafe programs in DSLs dynamically.

References

To be added.