Towards a Language for Defining Reusable Programming Language Components
(Project Paper, Extended Abstract)

Cas van der Rest[1] and Casper Bach Poulsen[2]

c.r.vanderrest@tudelft.nl[1] c.b.poulsen@tudelft.nl[2]
Delft University of Technology, Delft, The Netherlands

1 Introduction

Our goal is to build reusable programming language components from algebraic data types and pattern matching functions. Most functional programming languages, however, are less than ideal for this purpose, because they lack built-in solutions for the *Expression Problem* [13]: while adding functionality to existing data types is easily done by writing a new function, we cannot extend data definitions themselves without modifying existing code. For instance, consider the following implementation of a small expression language in Haskell:

```
data Expr = Lit Int | Div Expr Expr
eval :: Expr → Maybe Int
eval (Lit x)       = return x
eval (Div e1 e2) = do {x ← eval e1; y ← eval e2; safeDiv x y}
```

To extend this expression language with support for pretty-printing, we can simply define a new function $pretty :: Expr \to String$. But what if we want to extend it with a new construct corresponding to, for example, addition? Such a change would require us to go back and add a constructor *Add* to the definition of *Expr*, and extend all functions that match on *Expr* with new clauses accordingly.

This problem is amplified if we want to extend *Expr* with constructs that introduce new side-effects other than exceptions arising from division by zero. In that case, we also have to modify *eval*'s type signature, and potentially even the implementation of clauses for existing constructors. Clearly, if we intend *Expr* and *eval* as reusable components this is an undesirable situation.

We improve upon this state of affairs by introducing CS (working title), a functional meta-language for defining reusable programming language components. In CS, we can define components that describe part of a language's syntax, semantics or side effects, such that they can safely be composed into larger languages without requiring modification of existing code. The key features of CS that make this possible are (1) built-in support for per-case definitions of data types and pattern matching functions in the style of Data Types à la Carte [12], and (2) an effect system based on Plotkin and Pretnar's effect handlers [9] for the modular definition of side-effects.

CS is work in progress. There is a prototype implementation of an interpreter and interactive programming environment which we can us to define and run the examples from this abstract. We are, however, still in the process of developing and implementing a type system. In particular, we should statically prevent errors resulting from missing implementations of function clauses.

The name CS is an abbreviation of "Compositional Semantics". It is also the initials of Christopher Strachey, whose pioneering work [10] initiated the development of denotational semantics. In *Fundamental Concepts in Programming Languages* [11], Strachey wrote that "the urgent task in programming languages is to explore the field of semantic possibilities", and that we need to "recognize and isolate the central concepts" of programming languages. Today, five decades later, the words still ring true. The CS language aims to address this urgent task in programming languages, by supporting the definition of reusable (central) programming language concepts, via compositional denotation functions that map the syntax of programming languages to their meaning.

2 CS by Example

To showcase CS's design, we consider how to define the previous example as a reusable language component in CS.

Signatures and Modules The first step is to define a **signature** that announces the existence of an extensible data type *Expr*, and extensible function *eval*:

```
signature Eval (FX : Effects) where
  sort Expr : Set
  alg  eval  : Expr → {[FX] Int}
end
```

The braces ('{' and '}') in the type of *eval* indicate that it returns a suspended computation. Effects in CS happen eagerly, meaning that the side-effects of an expression occur *then and there* unless we suspend them. For *eval*, we do want suspension, leaving it up to the caller to decide when its effects take place.

The *Eval* signature has an effect row parameter, *FX*, describing which side effects may occur during evaluation. With the **alg** keyword, we allude to the initial algebra semantics for data types [5] on which CS's semantics for extensible types and functions is based. Indeed, we will see shortly that function clauses for *eval* are not implemented as regular functions, but as algebras instead.

We inhabit *Expr* and *eval* by defining **module**s that instantiate the *Eval* signature. We do this for the *Lit* and *Div* constructors:

```
module Lit : Eval where
  cons Lit : Int → Expr
  case eval (Lit n) = {n}
end
```

```
module Div : Eval where
  cons Div : Expr → Expr → Expr
  case eval (Div m1 m2) = {x ← m1; y ← m2; safeDiv x y}
```

Let us take a closer look at the implementation of *eval* in the module *Div*. There are two things worth noting here. First, we do not invoke *eval* recursively on the sub-expressions *m1* and *m2*. This is because we define function clauses as algebras, meaning that we assume that any recursive subtrees have already been replaced with the result of evaluating those subtrees. Second, the implementation uses the function *safeDiv* that guards against errors resulting from devision by zero. We find its implementation later on in the same module:

```
  fun safeDiv : Int → Int → [Abort] Int where
   | x 0 = abort !
   | x y = ...
end
```

The function *safeDiv* is annotated with the *Abort* effect, which supplies the abort operation, signalling abrupt termination. By invoking *safeDiv* in the defintion of *eval*, which from the definition of *Eval* has type $Expr \to [FX]\ Int$, we are implicitly imposing a constraint on the module parameter *FX* that it contains at least the *Abort* effect. In other words, whenever we import the module *Div* we better make sure that we instantiate *FX* with a row that has *Abort* in it.

We must say a few words about the braces ('{' and '}') that surround the implementation of *eval* for *Div*. Their purpose is to introduce a suspended computation. The opposite of suspension is enactment, which is denoted by postfixing with an exclamation mark (!). We see it in action in the definition of *safeDiv* (indeed, we abort immediately). Our use of braces is inspired by the similar language feature found in Frank [3].

Now, how do we use these modules to construct an interpreter for a language with integer literals and division? In CS, it is not necessary to explicitly compose constructors and clauses into data types and functions. Instead, the language manages this for us by automatically merging constructors and clauses whenever we import multiple instances of the same signature.

```
module Test where
  import Abort
         , Eval [Abort]
         , Lit, Div
  fun run : Expr → [Abort] Int where
   | e = eval e
   -- Evaluates to 3
  fun test : [Abort] Int
     = run (Div (Lit 6) (Lit 2))
end
```

We are allowed to invoke *eval* in the body of *run* here, because the sole constraint (imposed by importing *Div*) on its effect annotation of is that it contains *Abort*.

When importing the *Eval* signature we instantiated its effect row parameter with the singleton row $[\mathit{Abort}]$, which satisfies this constraint.

Effects and Handlers To use the *run* function, we must first invoke a handler for the *Abort* effect. To understand handlers, let us look at the module that implements the *Abort* effect together with its handler.

```
module Abort where
  import Prelude
  effect Abort where
   | abort : [Abort] a
  handler hAbort : [Abort | FX] a → [FX] (Maybe a) where
   | abort    k = Nothing
   | return x = Just x
end
```

With the **effect** keyword we declare a new effect together with its operations. Effect declarations are much like data type declarations, but instead of constructors they define the different ways in which we can construct effectful computations containing a particular effect.

We use the **handler** keyword to declare a handler for the *Abort* effect, *hAbort*, which removes it from the annotation of an effectful computation. The type of *hAbort* contains a free type variable (a) and a free row variable (FX), both of which are implicitly universally quantified, as is any free type or row variable. The result of handling the *Abort* effect is a *Maybe* value. *Maybe*, along with its constructors *Just* and *Nothing* is defined in the *Prelude* module.

Handlers must have a branch for each operation of the handled effect, plus a **return** branch that decorates pure values to match the handler's co-domain type. All branches corresponding to operations have an extra parameter that binds the *continuation*, representing the computation that succeeds the operation we are currently handling. By convention, we name this parameter k. In the abort case of *hAbort*, however, we ignore this continuation altogether, because the semantics of this operation should correspond to abrupt termination.

We use the continuation parameter in a more interesting way when defining a handler for a *State* effect:

```
module State (s : Set) where
  effect State where
   | get  :       [State] s
   | put  : s → [State] ()
  handler hState : [State | FX] a → s → [FX] (a×s) where
   | get          st k = k st st
   | (put st′)  st k = k () st′
   | return x st    = (x, st)
end
```

For both the get and put operations, we use the continuation parameter k to implement the corresponding branch in $hAbort$. The continuation expects a value whose type corresponds to the return type of the current operation, and produces a computation with the same type as the co-domain type of the handler. For the put operation, for example, this means that k is of type $() \to s \to [FX]\ (a \times s)$. The implementation of $hState$ for get and put then simply invokes k, using the current state as both the value and input state (get), or giving a unit value and using the given state st' as the input state (put).

2.1 Implementing Functions as a Reusable Effect

CS's effect system can describe much more sophisticated effects than *Abort* and *State*, as it permits fine-grained control over the semantics of operations that affect a program's control flow, even in the presence of other effects. To illustrate its expressiveness, we will now consider how to define function abstraction as a reusable effect, and implement two different handlers for this effect corresponding to a call-by-value and call-by-name semantics. We start by declaring the *Abstracting* effect and its operations:

```
effect Abstracting where
 | lam   : String → [Abstracting] Value → [Abstracting] Value
 | app   : Value  → Value               → [Abstracting] Value
 | var   : String                       → [Abstracting] Value
 | thunk : [Abstracting] Value          → [Abstracting] Value
```

The *Abstracting* effect has four operations, of which three correspond to the usual constructs of the λ-calculus. The thunk operation has no syntactical counterpart, but will be used for implementing a call-by-value and call-by-name evaluation strategy. *Value* is the type of values in our language; we will see shortly how it is defined.

When looking at the lam and thunk operations, we find that they both have parameters annotated with the *Abstracting* effect. This annotation indicates that they construct effectful computations from effectful computations, a pattern sometimes referred to as *higher-order effects*. Effectively, this means that any effects belonging to a value we wrap in a closure or thunk are postponed, leaving it up to the handler to decide when these take place.

Using the Abstracting effect To define a langue with function abstractions using the *Abstracting* effect, we define constructors *Abs*, *App*, and *Var* for *Expr*, and evaluate them by mapping onto the corresponding operation.

```
module Lambda : Eval where
  cons Abs : String → Expr → Expr
     |  App : Expr   → Expr → Expr
     |  Var : String        → Expr
  case eval (Abs x m)      = { lam x m }
  |    eval (App m1 m2) = { t ← thunk m2; app m1! t }
  |    eval (Var x)        = { var x }
end
```

Crucially, in the case for *Abs* we pass the effect-annotated body m, which has type { [*FX*] *Value* }, to the lam operation directly without extracting a pure value first. This prevents any effects in the body of a lambda from being enacted at the definition site, and instead leaves the decision of when these effects should take place to the used handler for the *Abstracting* effect. Similarly, in the case for *App*, we pass the function argument *m2* to the thunk operation directly, postponing any side-effects until we force the constructed thunk. We do, however, enact the side-effects of evaluating the function itself (i.e., *m1*), because the app operation expects its arguments to be a pure value.

We define the call-by-value and call-by-name handlers for *Abstracting* in a new module, that also defines the type of values for our language. Consequently, we adapt the *Eval* signature to use this value type in the signature for *eval*. To keep the exposition simple, we do not define *Value* as an extensible **sort**, but it is possible to do this in CS.

The values in this language are either numbers (*Num*), functions (*Clo*), or thunked computations (*Thunk*):

```
module HLambda (FX : Effect) where
  import Abstracting
  type Env = List (String × Value)
  data Value = Num Int
             | Clo String (Env → [Abort | FX] Value) Env
             | Thunk ([Abort | FX] Value)
   -- ... (handler for the Abstracting effect) ...
end
```

Call-by-value We are now ready to define a hander for the *Abstracting* effect that implements a call-by-value evaluation strategy. Figure 1 shows its implementation.

The **return** case is unremarkable: we simply ignore the environment nv and return the value v. The cases for lam and thunk are similar, as in both cases we do not enact the side-effects associated with the stored computation f, but instead wrap this computation in a *Closure* or *Thunk* which is passed to the continuation k. For variables, we resolve the identifier x in the environment and pass the result to the continuation.

A call-by-value semantics arises from the implementation of the app case. The highlights (e.g., $t!$) indicate where the thunk we constructed for the function argument in *eval* is forced. In this case, we force this argument thunk immedi-

```
handler hCBV  :  [Abstracting | FX] Value
                 → Env → [Abort | FX] Value where
 | (lam x f)                         nv k = k (Clo x f nv) nv
 | (app (Clo x f nv′) ( Thunk t ))   nv k = v′ ← f ((x, t! ) :: nv′)
                                          ; k v′ nv
 | (app _ _)                         _  _ = abort !
 | (var x)                           nv k = k (lookup nv x) nv
 | (thunk f)                         nv k = k (Thunk {f nv}) nv
 | return v                          nv   = v
```

Fig. 1. A Handler for the *Abstracting* effect, implementing a call-by-value semantics for function arguments. The gray highlights indicate where thunks constructed for function arguments are forced.

ately when encountering a function application, meaning that any side-effects of the argument take place *before* we evaluate the function body.

Call-by-name The handler in Figure 2 shows an implementation of a call-by-name semantics for the *Abstracting* effect. The only cases differences with the call-by-value handler in Figure 1 are the app and var cases.

In the case for app, we now put the argument thunk in the environment immediately, without forcing it first. Instead, in the case for var, we check if the variable we look up in the environment is a *Thunk*. If so, we force it and pass the resulting value to the continuation. In effect, this means that for a variable that binds an effectful computation, the associated side-effects take place every time we use that variable, but not until we reference it for the first time.

Example To illustrate the difference between *hCBV* (Figure 1) and *hCBN* (Figure 2), we combine the *Lambda* module with a module that uses the *State* effect. It defines expressions for reading and incrementing a single memory cell containing an integer:

```
module Mem : Eval where
  import State Int
  cons Incr   : Expr
  |    Recall : Expr
  case eval Incr   = {put (get ! +1); Num get!}
  |    eval Recall = get
end
```

When combining *Mem* and *Lambda*, we can observe the difference between a call-by-value and call-by-name evaluation strategy. Figure 3 shows an example of this.

```
handler hCBN  :  [Abstracting | FX] Value
              → Env → [Abort | FX] Value where
  | (lam x f)             nv k = k (Clo x f nv) nv
  | (app (Clo x f nv′) v) nv k = v′ ← f ((x, v) :: nv′)
                               ;  k v′ nv
  | (app _ _)             _  _ = abort !
  | (var x)               nv k = match lookup x nv with
                                   | ( Thunk t ) → k  t!  nv
                                   | v           → k v nv
                                 end
  | (thunk f)             nv k = k (Thunk {f nv}) nv
  | return v              nv   = v
```

Fig. 2. A Handler for the *Abstracting* effect, implementing a call-by-name semantics for function arguments. The gray highlights indicate where thunks constructed for function arguments are forced.

3 Outlook

CS is an ongoing research project. Here, we briefly summarize the current state, and some of the challenges that still remain.

Current state There is a prototype implementation of CS, consisting of an implementation of the operational semantics, a declarative type checker written in Statix [1], and an interactive environment through which we can interact with the language. The operational semantics is inspired by a recently-proposed flavor of effect handlers called *Latent Effects* [2], which unlike plain effects and handlers can describe many advanced control-flow mechanisms. The *Abstracting* effect and its different evaluation strategies are an example of how we can benefit from this extra expressivity. With this prototype, it is possible to define and run the examples shown in this abstract.

Static Semantics We are still in the process of developing a type system for CS. Our plan is to use row types for both effect annotations (e.g., à la Frank [3] and Koka [6]), and for typing extensible data types and functions. The motivation for the latter is that CS's static semantics should prevent problems arising from missing function clause declarations. Row types seem to be a good fit for this requirement, since they allow pattern matching functions to reflect in their type for which constructors they are defined. By assigning a row type to extensible functions, we statically make the necessary information available to check that they are not applied to an input for which there does not exist a corresponding **case** declaration. We draw inspiration from the ROSE [8] language, which applies row types to type extensible data types and records.

```
module Test where
  import Prelude
         , Abstracting
         , State Int
         , HLambdaCBV [State]
         , HLambdaCBN [State]
         , Lambda
         , Mem
  fun execCBV : Expr → (Maybe Value × Int) where
   | e = hState {hAbort {hCBV (eval e) [] }} 0
  fun execCBN : Expr → (Maybe Value × Int) where
   | e = hState {hAbort {hCBN (eval e) [] }} 0
  fun expr : Expr = App (Abs "x" Recall) Incr
   -- evaluates to (Just (Num 1) , 1)
  fun result1 : (Maybe Value × Int) = execCBV expr
   -- evaluates to (Just (Num 0) , 0)
  fun result2 : (Maybe Value × Int) = execCBN expr
end
```

Fig. 3. Examples of different outcomes when using a call-by-value or call-by-name evaluation strategy.

Core Language Parallel to developing CS, we are also working on developing a row-typed core language, which is intended as a minimal calculus to which we can desugar programs written in full CS. We base this core language on Rose [8], adapting it where necessary to encode (extensible) recursive data types and row-typed effects. Because the core language is much smaller than the surface language, it becomes more feasible to give a full formal specification of its semantics, and verify meta-theoretical properties such as type safety. The core language is still under development, but we hope to use it as a well-understood foundation for CS in the future. Of course, this will introduce additional challenges with respect to usability, such as how to provide decent error messages when type checking CS by going through the core language.

Semantics of extensible functions The current semantics of extensible functions is given by a catamorphism (fold) over the input type. This is a limiting factor when we try to implement traversals with a more complex recursive structure as an extensible function. To make CS's extensible functions in more expressive, we could switch to a richer model of extensible functions. For this we could explore, for example, more expressive recursion schemes [7], or *mixin algebras* [4].

References

1. van Antwerpen, H., Poulsen, C.B., Rouvoet, A., Visser, E.: Scopes as types. Proc. ACM Program. Lang. **2**(OOPSLA), 114:1–114:30 (2018). https://doi.org/10.1145/3276484, `https://doi.org/10.1145/3276484`
2. van den Berg, B., Schrijvers, T., Bach-Poulsen, C., Wu, N.: Latent effects for reusable language components: Extended version. CoRR **abs/2108.11155** (2021), `https://arxiv.org/abs/2108.11155`
3. Convent, L., Lindley, S., McBride, C., McLaughlin, C.: Doo bee doo bee doo. J. Funct. Program. **30**, e9 (2020). https://doi.org/10.1017/S0956796820000039, `https://doi.org/10.1017/S0956796820000039`
4. Delaware, B., d. S. Oliveira, B.C., Schrijvers, T.: Meta-theory à la carte. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 207–218. ACM (2013). https://doi.org/10.1145/2429069.2429094, `https://doi.org/10.1145/2429069.2429094`
5. Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Rocca, S.R.D. (ed.) Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4583, pp. 207–222. Springer (2007). https://doi.org/10.1007/978-3-540-73228-0_16, `https://doi.org/10.1007/978-3-540-73228-0_16`
6. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 486–499. ACM (2017). https://doi.org/10.1145/3009837.3009872, `https://doi.org/10.1145/3009837.3009872`
7. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings. Lecture Notes in Computer Science, vol. 523, pp. 124–144. Springer (1991). https://doi.org/10.1007/3540543961_7, `https://doi.org/10.1007/3540543961_7`
8. Morris, J.G., McKinna, J.: Abstracting extensible data types: or, rows by any other name. Proc. ACM Program. Lang. **3**(POPL), 12:1–12:28 (2019). https://doi.org/10.1145/3290325, `https://doi.org/10.1145/3290325`
9. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Log. Methods Comput. Sci. **9**(4) (2013). https://doi.org/10.2168/LMCS-9(4:23)2013, `https://doi.org/10.2168/LMCS-9(4:23)2013`
10. Strachey, C.: Towards a formal semantics (1966)
11. Strachey, C.S.: Fundamental concepts in programming languages. High. Order Symb. Comput. **13**(1/2), 11–49 (2000). https://doi.org/10.1023/A:1010000313106, `https://doi.org/10.1023/A:1010000313106`
12. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758, `https://doi.org/10.1017/S0956796808006758`
13. Wadler, P.: The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt` (1998), accessed: 2020-07-01