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

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

Abstract. Developing programming languages is a difficult task that requires a lot of time, effort, and expertise. Reusable programming language components make this task easier, by allowing language designers to grab off-the-shelf components for common language features. Modern functional programming languages, however, lack support for reuse of definitions, and thus language components defined using algebraic data types and pattern matching functions cannot be reused without modifying or copying existing code. To improve the situation, we introduce CS, a functional meta-language for developing reusable programming language components, that features built-in support for extensible data types and functions, as well as effects and handlers. In CS, we can define language components using algebraic data types and pattern matching functions, such that we can compose language components into larger languages and define new interpretations for existing components without modifying existing definitions.

1 Introduction

Developing programming languages is a difficult and time consuming task, which requires a lot of expertise from the language designer. One way to reduce the cost of language development is to build languages from *reusable programming components*, allowing language designers to grab off-the-shelf components for common language features. This approach has the potential to make language development both cheaper and more accesbile, while producing specifications that allow us to understand the semantics of language features independent from the language they are a part of. Modern functional programming languages, however, lack support for reuse of definitions, and as a result, language components built from algebraic data types and pattern matching functions cannot be reused without modifying or copying existing code. To illustrate the kind of language components we would like to define modularly, and where current functional programming languages fall short for this purpose, we consider the implementation of a tiny expression language and its interpreter in Haskell:

```
data Expr = Lit Int | Div Expr Expr
eval :: MonadFail m ⇒ Expr → m Int
eval (Lit x)        = Just x
eval (Div e1 e2) = do
  v1 ← eval e1
  v2 ← eval e2
  if v2 ≢ 0 then v1 ‘div‘ v2 else fail
```

The *Expr* data type declares an abstract syntax type with constructors for literals and division, and the function *eval* implements an interpreter for *Expr*. Importantly, *eval* needs to account for possible divisions by zero: evaluating *Div* (*Lit* 10) (*Lit* 0), for example, should safely evaluate to a result that indicates a failure, without crashing. For this reason *eval* does not produce an *Int* directly, but rather wraps its result in an abstract monad *m* that encapsulates the side effects of interpretation. In this case, we only assume that *m* is a member of the *MonadFail* typeclass. The *MonadFail* class hase one function, *fail*:

```
class MonadFail m where
  fail :: m a
```

We refer to functions, such as *fail*, that allow us to interact with an abstract monad as *operations*. We choose to factor language definitions this way, because it allows us to both define a completely new interpretation such as pretty printing or compilation for *Expr* by writing new functions *pretty* :: *Expr* → *String* or *compile* :: *Expr* → *m* [*Instr*], while also having the option to change the implementation of existing semantics, by supplying alternative implementations for the fail operation. We can summarize this approach to defining language components with the following pipeline:

$$\text{Syntax} \xrightarrow{\text{denotation}} \text{Operations} \xrightarrow{\text{implementation}} \text{Result}$$

That is, a *denotation* maps *syntax* to an appropriate domain. In the definition of this domain, we distinguish between the type of the resulting value, and the side effects of computing this result, which are encapsulated in an abstract monad. We interact with this abstract monad using *operations*, and thus to extract a result we must supply a monad that *implements* all required operations.

What if we want to extend this language? To add new constructors to the abstract syntax tree, we must extend the definition of *Expr*, and modify *all* functions that match on *Expr* accordingly. Furthermore, the new clauses for these constructors may impose additional requirements on *m* for which we would need to add more typeclass constraints, and any existing instantiations of *m* would need to be updated to ensure that they are still a member of all required typeclasses.

Clearly, for these reasons *Expr* and *eval* in their current form do not work very well as a reusable language component. We introduce CS, a functional meta-language for defining reusable programming language components. The goal of CS is to provide a language in which one can define language components by defining data types and pattern matching functions, like *Expr* and *eval*, in such a way that we compose the syntax, interpretations, and effects of a language component without affecting existing defintion. Importantly, we should also retain the possibility to add completely new interpretations for existing syntax by writing a new pattern matching function. In other words, CS should solve the *expression problem* [33].

We can summarize this with following concrete design goals. In CS, one should be able to

– extend existing abstract syntax types with new constructors without having to modify existing definitions,
– extend existing denotations with clauses for new constructors, and define new semantics for existing syntax by defining new denotations,
– define abstract effect operations, and use these operations to implement denotation clauses without having to worry about the operations needed by other clauses, and
– define implementations for effect operations that are independent from the implementations of other operations.

There exist abstractions, such as *Data Types à la Carte* [32] and *Algebraic Effects and Handlers* [25], that achieve the same goals. These provide the well-understood formalism on wich CS is built. CS then provides a convenient surface syntax for working with these abstractions that avoids the overhead that occurs when encoding them in a host language like Haskell.

CS is work in progress. There is a prototype implementation of an interpreter and interactive programming environment which we can use 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 detect errors resulting from missing implementations of function clauses.

The name CS is an abbreviation of "CompositionalSemantics". It is also the initials of Christopher Strachey, whose pioneering work [30] initiated the development of denotational semantics. In *Fundamental Concepts in Programming Languages* [31], 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

In this section, we give an example-driven introduction to CS's features.

2.1 Data Types and Functions

CS is a functional programming language, and comes equipped with algebraic data types and pattern matching functions. We declare a new inductive data type for natural numbers as follows:

```
data Nat = Zero | Suc Nat
```

We can write functions over inductive data types by pattern matching, using a "pipe" (|) symbol to separate clauses:

```
fun double : Nat → Nat where
 | Zero    ↦ Zero
 | (Suc n) ↦ Suc (Suc (double n))
```

Not all types are user-declared: CS also offers built-in types and syntax for integers, lists, tuples, and strings.

```
fun length : List a → Int where
 | []         ↦ 0
 | (_ :: xs)  ↦ 1 + length xs
fun zip : List a → List b → List (a ∗ b) where
 | []         _          ↦ []
 | (x :: xs) (y :: ys)   ↦ (x, y) :: zip xs ys
```

Both *length* and *zip* are polymorphic in the type of elements stored in the input list(s). Functions implicitly generalize over any free variables in their type signature.

2.2 Effects and Handlers

CS supports effectful programs by means of effects and handlers in the spirit of Plotkin and Pretnar [25], adapted to support higher-order operations. The key idea of the effects-and-handlers approach is to declare the syntax of effectful *operations*, and assign a semantics to these operations in a separate *handler*. Programs compute values and have side-effects, and operations act as the interface through which these side effects are triggered.

We declare a new effect *Fail* with a single operation fail in CS as follows:

```
effect Fail where
  fail : {[Fail] a}
```

Effects in CS are declared with the **effect** keyword, and we declare its operations by giving a list of GADT-style signatures. In this case, the fail operation is declared to have type $\{[\mathit{Fail}]\ a\}$. We enclose the type of fail in braces ($\{-\}$) to indicate that the name fail refers to a *suspended computation* (Section 2.3). Suspended computations are annotated with an *effect row*, enclosed in square brackets ([−]), denoting the side effects of running the computation. Invoking the fail operation has *Fail* as a side effect.

We can use the *Fail* effect to implement a safe division function that invokes fail if the divisor is zero.

```
fun safeDiv : Int → Int → [Fail] Int where
 | x 0 ↦ fail!
 | x y ↦ ...
```

The postfix exclamation mark to fail is necessary to force the suspended computation. Here, we want to refer to the "action" of failing, rather than the computation itself, following Frank's [10] separation between "being and doing". We elaborate on this distinction in Section 2.3.

A function's type signature must explicitly indicate its side-effects. In this case, we annotate the return type of *safeDiv* with the *Fail* effect to indicate that its implementation uses operations of this effect. Removing the annotation would make the above invocation of fail ill-typed. For functions that have no side-effects, we may omit its row annotation: $a \to b$ is synonymous to $a \to [\,]\ b$

Handlers discharge an effect from annotations by assigning a semantics to its operations. For the *Fail* effect, we can do this by encoding exceptions in the *Maybe* type.

```
data Maybe a = Just a | Nothing
handler hFail : {[Fail|e] a} → {[e] (Maybe a)} where
 | fail     k ↦ {Nothing}
 | return x ↦ {Just x}
```

The handler *hFail* takes a value annotated with the *Fail* effect, and produces a *Maybe* value annotated with the remaining effects e. All free effect row variables in a signature, like e, are implicitly generalized over. When defining a handler we must provide a clause for each operation of the handled effect. Additionally, we must write a **return** clause that lifts pure values into the domain that encodes the effect's semantics. Operation clauses have a *continuation parameter* (k), which captures the remainder of the program starting from the current operation. Handlers may use the continuation parameter to decide how execution should resume after the current operation is handled. For example, when handing the fail operation we terminate execution of the program by ignoring this continuation.

We use the continuation parameter in a different way when defining a handler for a *State* effect, where $s : Set$ is a parameter of the module in which we define the effect.

```
effect State where
 | get :        [State] s
 | put : s → [State] ()
handler hState : {[State|e] a} → s → {[e] (a * s)} where
 | get          st k ↦ k st st
```

```
  | (put st′)   st k ↦ k () st′
  | return x st      ↦ {(x, st)}
```

For both the get and put operations, we use the continuation parameter k to implement the corresponding branch in *hState*. 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 return type of the handler. For the put operation, for example, this means that k is of type $() \to s \to \{[e]\ (a * s)\}$. 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). Effectively, this means that after handling get or put, execution of the program resumes, respectively with the same state or an updated state st'.

2.3 Order of Evaluation, Suspension, and Enactment

Inspired by Frank [10], CS allows effectful computations to be used as if they were pure values, without having to sequence them. Sub-expressions in CS are evaluated from left to right, and the side-effects of computational sub-expressions are evaluated eagerly in that order. For example, consider the following program:

```
fun f : Int → [Fail] Int where
  | n ↦ fail! + n
```

Here, we use the expression fail! (whose type is instantiated to $[Fail]\ Int$) as the first argument to $+$, where a value of type *Int* is expected. This is fine, because side-effects that occur during evaluation of sub-terms are discharged to the surrounding context. That is, the side-effects of evaluating computational sub-terms in the definition of f become side-effects of f itself.

In practice, this means that function application in CS is not unlike programming in an *applicative style* in Haskell. For instance, when using the previously-defined handler *hFail*, which maps the *Fail* effect to a *Maybe*, we can informally understand the semantics of the CS program above as equivalent to the following Haskell program:

```
f :: Int → Maybe Int
f n = (+) <$> Nothing <*> pure n
```

Equivalently, we could write the following *monadic* program in Haskell, which makes the evaluation order explicit.

```
f :: Int → Maybe Int
f n = do x ← Nothing
         y ← pure n
         return (x + y)
```

CS's eager treatment of side-effects means that effectful computations are not first-class values, in the sense that we cannot refer to an effectful computation without triggering its side effects. To treat computations as first-class, we must explicitly *suspend* their effects using *braces*:

```
fun f' : Int → {[Fail] Int} where
| n ↦ {f n}
```

The function f' is no longer a function on *Int* that may fail, but instead a function from *Int* to a computation that returns and *Int*, but that could also fail. We indicate a suspended computation in types using braces ({/}), and construct a suspension at the term level using the same notation.

To *enact* the side-effects of a suspended computation, we postfix it with an exclamation mark (!). For example, the expression $(f'\ 0)!$ has type $[Fail]\ Int$, whereas the expression $(f'\ 0)$ has type $\{[Fail]\ Int\}$. We see the same distinction with operations declared using the **effect** keyword. When we write fail, we refer to the operation in a descriptive sense, and we can treat it like any other value without having to worry about its side-effects. When writing fail! , on the other hand, we are really performing the action of abruptly terminating: fail *is* and fail! *does*.

2.4 Modules and Imports

CS programs are organized using modules. Modules are delimited using the **module** and **end** keywords, and their definitions can be brought into scope elsewhere using the **import** keyword. All declarations—i.e., data types, functions, effects, and handlers—must occur inside a module.

```
module A where
  fun f : Int → Int where
   | n ↦ n + n
end
module B where
  import A
  fun g : Int → Int where
   | n ↦ f n
end
```

In addition to being an organizational tool, modules play a key role in defining and composing modular data types and functions.

2.5 Composable Data Types and Functions

In addition to plain algebraic data types and pattern matching functions, declared using the **data** and **fun** keywords, CS also supports case-by-case definitions of *extensible* data types and functions. In effect, CS provides a convenient surface syntax for working with DTC-style [32] definitions, which relies on an embedding of the initial algebra semantics [12] of data types to give a semantics to extensible and composable algebraic data types and functions, meaning that extensible functions have to correspond to a *fold* [19]. In CS, one can program with extensible data types and functions in the same familiar way as with their plain, non-extensible counterparts.

The module system plays an essential role in the definition of composable data types and functions. That is, modules can inhabit a *signature* that declares the extensible types and functions for which that module can give a partial definition. In a signature declaration, we use the keyword **sort** to declare an extensible data type, and the **alg** keyword to declare an extensible function, or *algebra*. By requiring extensible functions to be defined as algebras over the functor semantics of extensible data types, we enforce *by construction* that they correspond to a fold.

As an example, consider the following signature that declares an extensible data type *Expr*, which can be evaluated to an integer using *eval*.

```
signature Eval where
  sort Expr : Set
  alg  eval : Expr → Int
end
```

To give cases for *Expr* and *eval*, we define modules that inhabit the *Eval* signature.

```
module Lit : Eval where
  cons Lit : Int → Expr
  case eval (Lit x) ↦ x
end
module Add : Eval where
  cons Add : Expr → Expr → Expr
  case eval (Add x y) ↦ x + y
end
```

The **cons** keyword declares a new constructor for an extensible data type, where we declare any arguments by giving a GADT-style type signature. We declare clauses for functions that match on an extensible type using the **case** keyword. For every newly declared constructor of an extensible data type, we have an obligation to supply exactly one corresponding clause *for every extensible function that matches on that type*. CS has a coverage checker that checks whether modules indeed contain all necessary definitions, in order to rule out partiality resulting from missing patterns. For example, omitting the *eval* case from either the module *Lit* or *Add* above will result in a static error. Coverage is checked locally in modules, and preserved when composing signature instances.

In the definition of *eval* in the module *Add*, we see the implications of defining function clauses as algebras. We do not have direct control over recursive calls to *eval*. Instead, in **case** declarations, any recursive arguments to the matched constructor are replaced with the result of recursively invoking *eval* on them.

In this case, this implies that x and y do not refer to expressions. Rather, if we invoke *eval* on the expression *Add e1 e2*, in the corresponding **case** declaration, x and y are bound to *eval e1* and *eval e2* respectively. We could encode the same example in Haskell as follows, but to use *eval* on concrete expressions additionally requires explicit definitions of a type level fixpoint and fold operation. In CS, this encoding layer is hidden by the language.

```
data Add e = Add e e
eval :: Add Int → Int
eval (Add x y) = x + y
```

To compose signature instances we merely have to import them from the same location.

```
module Program where
  import Lit, Add
   -- Evaluates to 3
  fun test : Int = eval (Add (Lit 1) (Lit 2))
end
```

By importing both the *Lit* and *Add* modules, the names *Expr* and *eval* will refer to the composition of the constructors/clauses defined in the imported signature instances. Here, this means that we can construct and evaluate expressions that consist of both literals and addition. Furthermore, to add a new constructor into the mix, we can simply define a new module that instantiates the *Eval* signature, and add it to the import list.

To define an alternative interpretation for *Expr*, we declare a new signature. In order to reference the **sort** declaration for *Expr*, we must import the *Eval* signature.

```
signature Pretty where
  import Eval -- brings 'Expr' into scope
  alg pretty : Expr → String
end
```

We declare cases for *pretty* by instantiating the newly defined signature, adding **import** declarations to bring relevant **cons** declaration into scope.

```
module PrettyAdd : Pretty where
  import Add -- brings 'Add' into scope
  case pretty (Add s1 s2) = s1 ++ " + " ++ s2
end
```

3 Defining Reusable Language Components in CS

In this section, we demonstrate how to use the features of CS introduced in the previous section to define reusable language components. We work towards defining a reusable component for function abstraction, which can be composed with other constructs, and for which we can define alternative implementations. As an example, we will show that we can use the same component defining functions with both a call-by-value and call-by-name strategy.

3.1 A Signature for Reusable Components

The first step is to define an appropriate module signature. We follow the same setup as for the *Eval* signature in Section 2.5. That is, we declare an extensible sort *Expr*, together with an algebra *eval* that consumes values of type *Expr*. The result of evaluation is a *Value*, with potential side effects *e*. The side effects are still abstract in the signature definition. Later, when instantiating the *Eval* signature, we may impose constraints on *e* when implementing clauses of *eval* using operations of concrete effects.

```
signature Eval where
  sort Expr : Set
  alg  eval  : Expr → {[e] Value}
end
```

We will consider the precise definition of *Value* later in Section 3.3. For now, it is enough to know that it has a constructor *Num* : *Int* → *Value* that constructs a value from an integer literal.

3.2 A Language Component for Arithmetic Expressions

Let us start by defining instances of the *Eval* signature for the expression language from the introduction. First, we define a module for integer literals.

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

The corresponding clause for *eval* simply returns the value *n* stored inside the constructor. Because the interpreter expects that we return a suspended computation, we must wrap *n* in a suspension, even though it is a pure value. Enacting this suspension, however, does not trigger any side effects, and as such importing *Lit* imposes no constraints on the effect row *e*.

Next, we define a module *Div* that implements integer division.

```
module Div : Eval where
  cons Div : Expr → Expr → Expr
  case eval (Div m1 m2) = {safeDiv m1! m2! }
```

Looking at the implementation of *eval* in the module *Div* we notice two things. First, the recursive arguments to *Div* have been replaced by the result of calling *eval* on them, meaning that *m1* and *m2* are now *computations* with type

$\{[e]\ Int\}$, and hence we must use enactment before we can pass the result to *safeDiv*. Enacting these computations may trigger side effects, so the order in which sub-expressions are evaluated determines in which order these side effects occur in the case that expressions contain more than one enactment. Sub-expressions in CS are evaluated from left to right. Second, the implementation uses the function *safeDiv*, defined in Section 2.2, which guards against errors resulting from division by zero.

The function *safeDiv* is annotated with the *Fail* effect, which supplies the fail operation. By invoking *safeDiv* in the defintion of *eval*, which from the definition of *Eval* has type $Expr \to \{[e]\ Int\}$, we are imposing a constraint on the effect row e that it contains at least the *Fail* effect. In other words, whenever we import the module *Div* we have to make sure that we instantiate e with a row that has *Fail* in it. Consequently, before we can extract a value from any instantiation of *Eval* that includes *Div*, we must apply a handler for the *Fail* effect.

Since the interpreter now returns a *Value* instead of an *Int*, we must modify *safeDiv* accordingly. In practice this means that we must check if its arguments are constructed using the *Num* constructor before further processing the input. Since *safeDiv* already has *Fail* as a side effect, we can invoke the fail operation in case an argument was constructed using a different constructor than *Num*.

3.3 Implementing Functions as a Reusable Effect

CS's effect system can describe much more sophisticated effects than *Fail*. The effect system 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. Implementing function abstraction as an effect is especially challenging since execution of the function body is deferred until the function is applied. From a handler's perspective, this means that the function body and its side effect have to be postponed until a point beyond its own control, a pattern that is very difficult to capture using traditional algebraic effects.

We will see shortly how CS addresses this challenge. A key part of the solution is the ability to define *higher-order operations*: operations with arguments that are themselves effectful computations, leaving it up to the operation's handler to enact the side effects of higher-order arguments. The *Fun* effect, which implements function abstraction, has several higher-order operations.

```
effect Fun where
 | lam   : String → {[Fun] Value} → {[Fun] Value}
 | app   : Value  → Value         → {[Fun] Value}
 | var   : String                 → {[Fun] Value}
 | thunk : {[Fun] Value}          → {[Fun] Value}
```

The *Fun* effect defines four operations, three of which correspond to the usual constructs of the λ-calculus. The thunk operation has no counterpart in the

λ-calculus, and postpones evaluation of a computation. It is necessary for evaluation to support both a call-by-value and call-by-name evaluation strategy

When looking at the lam and thunk operations, we find that they both have parameters annotated with the *Fun* effect. This annotation indicates that they are higher-order parameters. By allowing higher-order parameters to operations, effects in CS do not correspond directly to algebraic effects. Instead, to give as semantics to effects in CS, we must use a flavor of effects that permits higher-order syntax, such as *Latent Effects* [7].

As a result, any effects of the computations stored in a closure or thunk are postponed, leaving it up to the handler to decide when these take place.

Using the Fun effect To define a langue with function abstractions using the *Fun* effect, we define an instance of the *Eval* signature that defines constructors *Abs*, *App*, and *Var* for *Expr*. We define *eval* for these constructors 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)    = app m1! (thunk m2)!
   |   eval (Var x)        = var x
end
```

Crucially, in the case for *Abs* we pass the effect-annotated body m, which has type $\{[e]\ Value\}$, 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 *Fun* 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. The precise moment at which we will force the thunk constructed for function arguments will depend on whether we employ a call-by-value or call-by-name strategy. We must, 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 *Fun* in a new module, which also defines the type of values, *Value*, for our language. To keep the exposition simple, we do not define *Value* as an extensible **sort**, but it is possible to do so in CS.

Values in this language are either numbers (*Num*), function closures (*Clo*), or thunked computations (*Thunk*). We define the type of values together in the same module as the handler(s) for the *Fun* effect. This module is parameterized over an effect row e, that denotes the *remaining effects* that are left after handling the *Fun* effect. In this case, e is a module parameter to express that

```
handler hCBV  : {[Fun|e] Value}
              → Env → {[Fail|e] Value} where
 | (lam x f)                          nv k ↦ k (Clo x f nv) nv
 | (app (Clo x f nv′) ( Thunk t ))    nv k ↦ k (f ((x, t! ) :: nv′))! nv
 | (app _ _)                          _  _ ↦ {fail! }
 | (var x)                            nv k ↦ k (lookup nv x)! nv
 | (thunk f)                          nv k ↦ k (Thunk {f nv}) nv
 | return v                           nv   ↦ {v}
```

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

the remaining effects in the handlers that we will define coincide with the effect annotations of the computations stored in the *Clo* and *Thunk* constructors, allowing us to run these computations in the handler.

```
module HLambda (e : Effects) where
  import Fun
  type Env   = List (String ∗ Value)
  data Value = Num Int
             | Clo String (Env → {[Fail|e] Value}) Env
             | Thunk ({[Fail|e] Value})
   -- ... (handlers for the Fun effect) ...
end
```

Call-by-value We are now ready to define a handler for the *Fun* 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 immediately 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 *Fun* effect. The only case that differ from the call-by-value handler in Figure 1 are the app and var cases.

```
handler hCBN  :  {[Fun|e] Value}
              → Env → {[Fail|e] Value} where
  | (lam x f)              nv k ↦ k (Clo x f nv) nv
  | (app (Clo x f nv′) v)  nv k ↦ k (f ((x, v) :: nv′))! nv
  | (app _ _)              _  _ ↦ {fail! }
  | (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}
```

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

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.

3.4 Example Usage

To illustrate how to use the reusable components defined in this section, and the difference between the semantics implemented by *hCBV* (Figure 1) and *hCBN* (Figure 2), we combine the *Lambda* module with the modules for *Div* and *Lit*. Figure 3 shows the example.

When importing the modules *HLambdaCBV* and *HLambdaCBN*, we pass an explicit effect row that corresponds to the effects that remain after handling the *Fun* effect. Because we handle *Fun* after handling the *Fail* effect introduced by *Div*, we pass the empty row. To evaluate expressions, we have to invoke *hFail* twice: first to handle the instance of the *Fail* effect introduced by *eval* for the *Div* constructor, and later to handle the *Fail* instance introduced by applying *hCBV*/*hCBN*. Consequently, the result of evaluating is a nested *Maybe*, where the inner instance indicates errors resulting from division by zero, and the outer instance errors thrown by the handler. Evaluating *result1* and *result2* shows the difference between using the call-by-value and call-by-name semantics for functions.

```
module Test where
  import Prelude
         , Fun
         , Fail
         , HLambdaCBV [ ]
         , HLambdaCBN [ ]
         , Lambda
         , Lit
         , Div
  fun execCBV : Expr → Maybe (Maybe Value) where
   | e ↦ (hFail (hCBV (hFail (eval e)) [ ]))!
  fun execCBN : Expr → Maybe (Maybe Value) where
   | e ↦ (hFail (hCBN (hFail (eval e)) [ ]))!
  fun expr : Expr = App (Abs "x" (Lit 10)) (Div (Lit 5) (Lit 0))
   -- evaluates to Just Nothing
  fun result1 : Maybe (Maybe Value) = execCBV expr
   -- evaluates to Just (Just (Num 10))
  fun result2 : Maybe (Maybe Value) = execCBN expr
end
```

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

4 Related Work

Effect Semantics. Monads, originally introduced by Moggi [20] have long been the dominant approach to modelling programs with side effects. They are, however, famously hard to compose, leading to the development of *monad transformers* [17] as a technique for building monads from individual definitions of effects. *Algebraic effects* [24] provide a more structured approach towards this goal, where an effect is specified in terms of the *operations* that we can use to interact with it. The behaviour of these operations is governed by a set of equations that specify its well-behavedness. Later, Plotkin and Pretnar [25] extended the approach with *handlers*, which define interpretations of effectful operations by defining a homomorphism from a *free model* that trivially inhabits the equational theory (i.e., syntax) to a programmer-defined domain, making the approach attractive for implementing effects as well. Perhaps the most well-known implementation of algebraic effects and handlers is the *free monad* [14], and this implementation is often taken as the semantic foundation of languages with support for effect handlers. Schrijvers et al. [29] showed that algebraic effects implemented using the free monad correspond to a sub-class of monad-transformers. The algebraic effects and handlers approach provides a solid formal framework for understanding effectful programs in which we intend to ground CS' semantics of effects and handlers.

A crucial difference between CS' effects and handlers, and the original formulation by Plotkin and Pretnar [25], is the support for *higher-order operations*. Although it is possible to implement such operations in algebraic effects by defining them as handlers, this breaches the seperation between the syntax and semantics of effects that underpins CS' design. *Scoped Effects* [35] were proposed as an alternative flavor of algebraic effects that supports higher-order syntax, recovering a separation between the syntax semantics of effects for higher-order operations. In subsequent work, Piróg et al. [23] adapted the categorical formulation of algebraic effects to give Scoped Effects a similar formal underpinning. Unfortunately, Scoped Effects is not suitable out-of-the-box as a model for effects and handlers in CS, because it cannot readily capture operations that arbitrarily postpone the execution of their higher-order arguments, such as lam. *Latent effects* were developed by Van den Berg et al. [7] as a refinement of scoped effects that solves this issue. Key to their approach is a *latent effect functor*, which explicitly tracks semantic residue of previously-installed handlers, allowing for a more fine-grained specification of the types of the computational arguments of operations. With Latent Effects, it is possible to capture function abstraction as a higher-order operation. It remains future work to formulate a precise model of effectful computation for CS, and to establish if and how CS' effect handlers correspond to Latent Effects.

Implementations of Algebraic Effects and Handlers. There are many languages with support for algebraic effects and handlers. Perhaps the most mature is Koka [15], which features a Hindley/Milner-style row-polymorphic type system. While we borrow from Frank [10] a CBPV-inspired [16] distinction between computations and values, Koka is purely call-by-value, and only functions can be effectful. Frank [10], on the other hand, does maintain this distinction between values and computations. Its type system relies on an *ambient ability* and implicit row polymorphism to approximate effects. Handlers are not first-class constructs in Frank. Instead, functions may adjust the ambient ability of their arguments by specifying the behaviour of operations. This provides some additional flexibility over built-in handers, for example by permitting *multihandlers* that handle multiple effects at once. Both Koka and Frank lack native support for higher order effects, thus higher-order operations must be encoded in terms of handlers. This means that it is not possible to define higher order operations while maintaining the aforementioned distinction between the syntax and semantics of effects.

Eff [6] is a functional language with support for algebraic effects and handlers, with the possibility to dynamically generate new operations and effects. In later work, Bauer and Pretnar [5] developed a type-and-effect system for Eff, together with an inference algorithm [26]. The language Links [18] employs row-typed algebraic effects in the context of database programming. Their system is based on System F extended with effect rows and row polymorphism, and limits effectful computations to functions similar to Koka. Importantly, their system tracks effects using Rémy-style rows [27], maintaining so-called *presence types* that can additionally express an effect's absence from a computation. Brachthäuser et al. [9] presented *Effekt* as a more practical implementation of effects and handlers, using *capability based* type system where effect types express a set of capabilities that a computation requires from its context.

Semantics of Composable Data Types and Functions. We give a semantics to extensible data types and functions in CS using the initial algebra semantics [12] of an underlying signature functor. *Data Types à la Carte* (DTC) [32] solves the expression problem in Haskell by embedding this semantics into the host language. In later work, Bahr [2] and Bahr and Hvitved [3,4] extended the approach to improve its expressiveness and flexibility.

DTC, like any approach that relies on initial algebra semantics, limits the modular definition of functions to functions that correspond to a *fold* over the input data. While this may seem restrictive, in practice more complicated traversals can often be encoded as a fold, such as paramorphisms [19] or some classes of attribute grammars [13]. While CS currently only has syntax for plain algebras and folds, we plan to extend the syntax for working with extensible data types and functions to accomodate a wider range of traversals in the future.

Row Types. While a concrete design for CS' type system is still emerging, we anticipate that it will make heavy use of *row types*, both for tracking effects and typing extensible types and functions. While to the best of our knowledge no type system exists with this combination of features, all the ingredients are there in the literature. Originally, row types were incepted as a means to model inheritance in object-oriented languages [34,27], and later extensible records [8,11]. More recently, they also gained popularity in the form of row-based effect systems with the development of languages such as Koka [15] and Links [18]. Their use for typing extensible algebraic data types and pattern matching functions is less well-studied. For the most part, row types in this context exist implicitly as part of encoding techniques such as DTC [32], where we can view the use of signature functors and functor co-products as an embedding of row-typed extensible variants in the host language's type system. Various refinements of DTC [21,28,2] make this connection more explicit by using type-level lists to track the composition of extensible data. A notable exception is the Rose [22] language, which has a row-based type system with built-in support for extensible variants and pattern matching function.

5 Future Work

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

While we can implement the examples from this paper in the current prototype implementation of CS, the language still lacks a complete specification. As such, the immediate next steps are to develop specifications of the type system and operational semantics. This requires us to address several open research questions, such as giving a semantics to the flavor of higher-order effects used by CS, and applying row types to type CS' extensible data types and functions. While the Rose language supports extensible variants, this support is limited to non-recursive types. For CS, we would need to adapt their type sytem to support recursive extensible data types as well. Designing a small core calculus into which CS can be translated could be potential way to explore these questions, making a formalization of the language in a proof assistant more attainable, by formalizing the core language. Further down the line, we also intend to explore a denotational model for effect handlers in CS, giving the language a more solid formal foundation, similar to existing programming languages based on algebraic effects and handlers.

In the future, we also hope to enforce stronger properties about specifications defined in CS through the language's type system. The prime example are *intrinsically-typed definitional interpreters* [1], which specify a language's operational semantics such that it is *type sound by construction*.

6 Conclusion

Reusable programming language components have the potential to significantly reduce the amount of time, effort, and expertise needed for developing programming languages. In this paper, we presented CS, a functional meta-language for defining reusable programming language components. CS enables the defintion of reusable language components using algebraic data types and pattern matching functions, by supporting *extensible data types and functions*, which are defined on a case-by-case basis. Additionally, CS features built-in support for *effects and handlers* for defining the side effects of a language. The flavor of effects and handlers implemented by CS supports *higher-order* operations, and can be used to define features that affect a program's control flow, such as function abstraction, as a reusable effect. We illustrated how these features can be used for developing reusable programming language components by defining a component for function abstraction, which can be composed with other language components and evaluated using both a call-by-value and call-by-name strategy.

References

1. Augustsson, L., Carlsson, M.: An exercise in dependent types: A well-typed interpreter. In: In Workshop on Dependent Types in Programming, Gothenburg (1999)
2. Bahr, P.: Composing and decomposing data types: a closed type families implementation of data types à la carte. In: Magalhães, J.P., Rompf, T. (eds.) Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014. pp. 71–82. ACM (2014), https://doi.org/10.1145/2633628.2633635
3. Bahr, P., Hvitved, T.: Compositional data types. In: Järvi, J., Mu, S. (eds.) Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 83–94. ACM (2011), https://doi.org/10.1145/2036918.2036930

4. Bahr, P., Hvitved, T.: Parametric compositional data types. In: Chapman, J., Levy, P.B. (eds.) Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012. EPTCS, vol. 76, pp. 3–24 (2012), `https://doi.org/10.4204/EPTCS.76.3`
5. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. Log. Methods Comput. Sci. **10**(4) (2014), `https://doi.org/10.2168/LMCS-10(4:9)2014`
6. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebraic Methods Program. **84**(1), 108–123 (2015), `https://doi.org/10.1016/j.jlamp.2014.02.001`
7. van den Berg, B., Schrijvers, T., Poulsen, C.B., Wu, N.: Latent effects for reusable language components. In: Oh, H. (ed.) Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13008, pp. 182–201. Springer (2021), `https://doi.org/10.1007/978-3-030-89051-3_11`
8. Blume, M., Acar, U.A., Chae, W.: Extensible programming with first-class cases. In: Reppy, J.H., Lawall, J.L. (eds.) Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006. pp. 239–250. ACM (2006), `https://doi.org/10.1145/1159803.1159836`
9. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effects as capabilities: effect handlers and lightweight effect polymorphism. Proc. ACM Program. Lang. **4**(OOPSLA), 126:1–126:30 (2020), `https://doi.org/10.1145/3428194`
10. 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`
11. Gaster, B.R., Jones, M.P.: A polymorphic type system for extensible records and variants. Tech. rep., Citeseer (1996)
12. 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`
13. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings. Lecture Notes in Computer Science, vol. 274, pp. 154–173. Springer (1987), `https://doi.org/10.1007/3-540-18317-5_10`
14. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013. pp. 145–158. ACM (2013), `https://doi.org/10.1145/2500365.2500590`
15. 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`
16. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semantics Structures in Computation, vol. 2. Springer (2004)
17. Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San

Francisco, California, USA, January 23-25, 1995. pp. 333–343. ACM Press (1995), https://doi.org/10.1145/199448.199528
18. Lindley, S., Cheney, J.: Row-based effect types for database integration. In: Pierce, B.C. (ed.) Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012. pp. 91–102. ACM (2012), https://doi.org/10.1145/2103786.2103798
19. 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
20. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991), https://doi.org/10.1016/0890-5401(91)90052-4
21. Morris, J.G.: Variations on variants. In: Lippmeier, B. (ed.) Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015. pp. 71–81. ACM (2015), https://doi.org/10.1145/2804302.2804320
22. 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
23. Piróg, M., Schrijvers, T., Wu, N., Jaskelioff, M.: Syntax and semantics for operations with scopes. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 809–818. ACM (2018), https://doi.org/10.1145/3209108.3209166
24. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Appl. Categorical Struct. **11**(1), 69–94 (2003), https://doi.org/10.1023/A:1023064908962
25. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: Castagna, G. (ed.) Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5502, pp. 80–94. Springer (2009), https://doi.org/10.1007/978-3-642-00590-9_7
26. Pretnar, M.: Inferring algebraic effects. Log. Methods Comput. Sci. **10**(3) (2014), https://doi.org/10.2168/LMCS-10(3:21)2014
27. Rémy, D.: Typechecking records and variants in a natural extension of ML. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989. pp. 77–88. ACM Press (1989), https://doi.org/10.1145/75277.75284
28. d. S. Oliveira, B.C., Mu, S., You, S.: Modular reifiable matching: a list-of-functors approach to two-level types. In: Lippmeier, B. (ed.) Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015. pp. 82–93. ACM (2015), https://doi.org/10.1145/2804302.2804315
29. Schrijvers, T., Piróg, M., Wu, N., Jaskelioff, M.: Monad transformers and modular algebraic effects: what binds them together. In: Eisenberg, R.A. (ed.) Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019. pp. 98–113. ACM (2019), https://doi.org/10.1145/3331545.3342595

30. Strachey, C.: Towards a formal semantics (1966)
31. 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`
32. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008), `https://doi.org/10.1017/S0956796808006758`
33. Wadler, P.: The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt` (1998), accessed: 2022-04-04
34. Wand, M.: Type inference for record concatenation and multiple inheritance. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989. pp. 92–97. IEEE Computer Society (1989), `https://doi.org/10.1109/LICS.1989.39162`
35. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Swierstra, W. (ed.) Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014. pp. 1–12. ACM (2014), `https://doi.org/10.1145/2633357.2633358`