Extending Meta-Casanova with Modules and Type Operators

Douwe van Gijn

supervised by
Dr. Giuseppe Maggiore

2016-03-13

Abstract

Meta-Casanova is a functional, declarative, and simply-typed language designed to write compilers. In this paper, we extend the language with type constructors and a module system to allow generic abstractions like parser monads and monad transformers. These abstractions are invaluable to compiler-writers as they help manage the complexity of modern-day compilers.

Contents

1 Introduction

A strong type system is essential for a functional language. It enables powerful abstractions like monad transformers to happen at compile-time, making it ideal for a fast compiler language.

Because the focus of Meta-Casanova is on writing compilers, we begin with a brief overview of monadic parsing. We show that parser monads can be constructed from more elementary monads, and that there is a simple way to combine these monads into a parser monad. We then show that we need a stronger typesystem than what is currently supported by Meta-Casanova and that we need to extend it with Modules and Type Operators. Furthermore, we will discuss in detail the design of this type system.

The focus of this paper is about the type system and the modules we have developed during the minor. To learn more about the syntax and semantics of Meta-Casanova, refer to the paper by Jarno Holstein[1]. To learn about the evolution of the standard library of Meta-Casanova and its applications, refer to the paper by Louis van der Burg[2].

2 The Problem

The goal of this paper is to extend Meta-Casanova's typesystem to the point where it can handle monad-transformers. In this section we present an extention of the type system of Meta-Casanova. We start by giving an overview of monadic parsers, then we move on to monad transformers, and we conclude with a look at what typesystem we need.

Monadic Parsing

Parser monads are a useful tool for writing compilers. They are not only useful in the lexing stage, as they can be used for parsing intermediate representations. Such intermediate representations appear between different sections of the compiler and are often tree-like. An example would be the typechecker, which has to parse the abstract syntax tree from the front-end.

The best aspect about monadic parsing is that it is composable. It is possible to combine multiple parsers into a single parser. For example: there might be a parser (`digit`) that can parse a single digit, and a parser combinator (`repeat`) that can apply a parser multiple times until it fails. It is then trivial to combine these into a new parser that parses multiple digits.

```
let digits = repeat digit
```

A parser is a function that takes an input stream and a context and returns a result. The input stream is a sequence of characters that has yet to be parsed. The context is the state that is kept and modified during the parsing. The result is either the succesful outcome of the parser, or an error message. A generic implementation may look like this.

```
type Parser<'char,'ctxt,'result> =
  List<'char> * 'ctxt
    -> Result<'char,'ctxt,'result>
```

The Result contains –if the parser was successful– the value returned by the parser, the modified context, and the new input stream position.

```
type Result<'char,'ctxt,'result> =
  | Done  of 'result*List<'char>*'ctxt
  | Error of string
```

Monad Transformers

It may not be obvious, but the parser monad can be constructed from more primitive monads. The input stream and the context are represented as a state-monad, and the result is an either-monad.

```
type Parser<'char,'ctxt,'result> =
  State<List<'char>*'ctxt,
        Either<'result,'error>>
```

Combining monads by hand is tedious and non-trivial. Luckily there is a way to automate the process: monad transformers.[3, Chapter 18]

Monad transformers are type operators (type-level functions) that takes a monad, and returns a combined monad. For example: if we pass the identity monad to the either monad transformer, we get an either monad. If we then pass the result monad to a state monad transformer, we get a result monad that can hold state. If we assign a list of input characters and a context to the state, we have constructed a generic parser monad.

In a hypothetical language where we could write this, it woult look like this:

```
type ParserM = StateT (ResultT IdM)
```

Where as writing one by hand would require the type definitions, as well as the implementation `return` and `>>=`.

```
type Result<'char,'ctxt,'result> =
  | Done  of 'result*List<'char>*'ctxt
  | Error of string
```

```
type Parser<'char,'ctxt,'result> =
  List<'char> * 'ctxt
    -> Result<'char,'ctxt,'result>

let return res =
  fun (chars,ctxt) ->
    Done(res,chars,ctxt)

let (>>=) p k =
  fun (chars,ctxt) ->
    match p (chars,ctxt) with
    | Error(p) ->
        Error(p)
    | Done(res,chars',ctxt') ->
        k res (chars',ctxt')
```

Towards F_ω

In order to have type operators, we need higher-order polymorphism. The formal name for a type system that supports higher-order polymorphism, System-F_ω.[4, Chapter 30] Now that it is clear what is needed, we can show what language features we need to add.

3 The Idea

In the previous section we have seen what kind of abstractions we want. In this section, we show the language features needed to allow those abstractions. We will first show that kinds ensure sound type construction, then we will show a powerful module system, and we will conclude with the inheritance model of these modules.

Kinds

Kinds are to types as types are to terms.[4, Chapter 30] They are “one level up” from types. The most used kind is $*$ (pronounced “type”) and represents all proper types. Type operators are indicated with $\Rightarrow$ to differentiate them from type-signatures ($\rightarrow$). For example a few types with their respective kinds:

```
int             *
float           *
pair            * ⇒ * ⇒ *
pair int        * ⇒ *
pair float int  *
pair pair       error
```

`pair pair` is not a valid type, because type `pair` has kind `*⇒*⇒*`. It only accepts two types with kind `*`. Kinds are necessary to prevent these nonsensical types from typechecking.

Kinds –unlike types– have no run-time representation. They don't need it, they are only executed at compile-time to check the validity of the types.

This property of kinds can be used to compute arbitrary expressions at compile-time. In Meta-Casanova, these expressions are called `TypeFuncs`, as they operate on types.

Modules

Modules in Meta-Casanova are a compile-time mechanism that allows the definition of interfaces. It re-uses `TypeFuncs` to declare new kinds.

To illustrate modules, let's implement a generic monad. We first define the monad interface.

```
TypeFunc "Monad" ⇒ (* ⇒ *) ⇒ Module
Monad 'M ⇒ Module {
  Func 'a →">>=" → ('a →'M 'b) →'M 'b
  Func "return" →'a →'M 'a
}
```

Now that we have the interface, let's instantiate an identity monad. To use the interface, we first declare a type that fits the first argument. It has kind `(* ⇒ *)` and represents the monad constructor. In case of the identity monad this is trivial.

```
TypeFunc "IdCons" ⇒ * ⇒ *
IdCons 'a ⇒ 'a
```

The first line declares that we have a new type operator Id that takes a type and returns a type. The second line implements it.

To instantiate the interface we create a type operator that takes no arguments and gives us a Monad.

```
TypeFunc "id" ⇒ Monad
id ⇒ Monad IdCons {
  x >>= k → k x
  return x → x
}
```

We have now successfully implemented a monad that does nothing. While it might look useless, it is actually used quite often.

In section 2 under **monad transformers**, we showed that the identity monad can be used to get monads out of monad transformers. For example: when you pass the identity monad to a state monad transformer, you get a state monad.

In fact, when using monad transformers, the identity monad forms the foundation of every monad. At the end of the chain of monad transformers, there is always an identity monad where all those transformations are applied to.

Inheritance

Modules can inherit from each other. This is done with the `Inherit` keyword. If –for instance– we want to extend our monad module with Zero, we can define a new module MonadZero that inherits from Monad.

```
TypeFunc "Zero" ⇒ Module
Zero ⇒ Module {
  Func "zero" → 'a
}

TypeFunc "MonadZero" ⇒ (*⇒*) ⇒ Module
MonadZero 'M ⇒ Module {
  Inherit Monad 'M
  Inherit Zero
}
```

This way, we have easily extended Monad to have a zero. Now MonadZero can behave like a monad, a 'something' with a zero, and a monad with a zero. In case of monads, having a zero often useful. In F# for example, defining a zero for a monad means you can omit the else-branch in an if-statement.

4 Details

4.1 Meta-Casanova

Meta-Casanova is a functional, declarative language. It allows for multiple implementations of functions called rules. Rules may fail and the program will continue with the successful ones.

Multiple rules may match at any given time. If this happens, the program execution splits into multiple branches; each branch following one rule. If none of the rules match, the branch dies off.

This mechanism effectively implements lookahead-behavior in programs, and is therefore useful for writing parsers.

Data

`Data` declarations declare a two-way many-to-1 relation between types. This two-way relationships makes `Data` an alias.

```
Data "nil" → list 'a
Data 'a → "::" → list 'a → list 'a
```

In this example, the list type is declared with two constructors. They specify that a lists can be constructed in two ways: with `nil` and with `::` surrounded with a term of type `'a`, and a term of type `list 'a`.

Conversely, they also specify that an list can be destructed in two ways. The programmer will assert which destructor is expected, and the rule fails if the destructor does not match. An example of this is shown later, in subsection “Funcs”.

Additionally, constructors may be manipulated and partially applied like functions. This allows for greater flexibility at the cost that function and constructor names need to be unique in their namespace.

Polymorphism

Polymorphic data structures are supported with the **`is`** keyword.

```
Data "error" → string → failableList 'a
failableList 'a is list 'a
```

This means every constructor of the `list` is also a valid constructor of `failableList`, but not vice-versa.

Funcs

Func declarations specify a new function and its type.

```
Func "length" → list 'a → int
```

As with constructors, functions may be freely manipulated and partially applied, and have the restriction that their name must be unique in their namespace.

Rules

Meta-Casanova uses a syntax similar to that of natural deduction. For each Func declaration, there are one or more rules that define it.

```
---------------
length nil -> 0

length xs -> res
---------------------
length x::xs -> 1+res
```

A rule is comprised of a line with below it on the left of the arrow the input, and on the right the output. The statements above the horizontal line are called premises. They can be assignments like in the example above, or conditionals like `a=b` or `c<d`.

We can now call the function `length` with an example list:

```
  1::(2::nil) -> x
  length x    -> res
```

The first premise constructs a list called “x”, and the second statement calls length with that list. The program will execute as follows:

```
length 1::(2::nil)
    ~~nil~~
    x::xs → 1+(length 2::nil)
        ~~nil~~
        x::xs → 1+(length nil)
            nil → 0
            ~~x::xs~~
```

After which the function stops calling itself and starts accumulating the result on the way down.

```
        1+0 → 1
    1+1 → 2
2
```

After which it tells us correctly that the length of the list 1::(2::nil) is indeed 2.

4.2 TypeFunc Details

TypeFuncs were designed to be consistent with the already existing Funcs. This has a few implications:

TypeFuncs curry

like Funcs, TypeFuncs curry.

Currying enables partially application. For example, imagine we have a state monad.

```
TypeFunc "StateMonad" ⇒ * ⇒ * ⇒ *
```

Where the first argument represents the state type and the second the return type. This means you can define a new state monad that already has its state-type specified by writing `Monad(StateMonad int)`.

It also gives the compiler a uniform way to handle type and function application, so this feature increases orthogonality while not increasing compiler complexity.

Double arrows

We choose double arrows for TypeFuncs to clearly differentiate them from function application(→). This distinction is to separate run-time from compile-time.

This is unlike Haskell, it uses single arrows (→). Haskell can do this because it does not have the explicit run-time/compile-time distinction Meta-Casanova has.

Typefunc premises

Typefuncs can have premises, just like rules. This means that TypeFuncs can behave just like rules and can compute arbitrary expressions at compile-time. The only limit is that values can't be passed at compile-time, as that would require dependent types.

TypeAliases

`TypeAlias` declarations are the mechanism for declaring type-level two-way relationships between kinds. This is analogous to the relationship `Data` declarations describe between types. TypeAliases enable kind-level pattern-matching in TypeFuncs, essential for making them behave like Funcs.

4.3 Module Details

Modules were designed to be as flexible as possible, to allow them to model as great a range of abstractions as possible. This means minimizing restrictions, and maximizing orthogonality. This philosophy manifests itself in the following ways:

Scope contents

Scopes mainly consist of `Func` declarations, but can be used for everything. They are equivalent to top-level scope[1]. We choose not to artificially limit the scope to maximize expressivity.

Modules declare new kinds

The simplest solution was to reuse the `TypeFunc` system for compile-time tasks. Kinds are a compile-time mechanism, and TypeFuncs are used for all compile-time tasks.

Inheritance propegation

Inherit statements inherit recursively. If C inherits from B, and B inherits from A, then C inherits both B and A. This is different from the way imports work. Import statements don't import recursively. This is to prevent the namespace getting cluttered with dependencies.

Module premises

Modules can have premises just like Funcs and TypeFuncs. This means you can arbitrarily manipulate the input arguments. Again the restriction here is the inability to pass term-level

[1]Except for `import` statements, those are not allowed in modules.

arguments to modules, as that would require dependent types.

Inheritance vs composition

Meta-Casanova uses an inheritance-based approach as opposed to a composition-based approach.

In the inheritance-based approach, kinds inherit from each other and the users of the kinds only accept one specific kind. In the case of MonadZero, MonadZero inherits from Monad and Zero, and Funcs written for MonadZero only accept MonadZero or derivatives.

In the composition-based approach, kinds do not inherit from each other, and the users of the kinds accept a range of kinds. In the case of MonadZero, there would only be a Monad module and a Zero module and the Funcs that needed both would specify that in their signature.

Both strategies have their merits and are used in programming languages successfully. The advantage of a composition-based approach is that you only have to specify the building blocks, and don't have to specify all the combinations. The advantage of a inheritance-based approach is that you can add special behavior to your modules.

Because flexibility is the main concern, Meta-Casanova uses inheritance.

5 Conclusions

In this paper we described the extensions to Meta-Casanova that add modules and type operators. The extensions pave the way for modular libraries with strong abstractions and leaves room for dependent types in the future.

Related Work

To learn more about the syntax and semantics of Meta-Casanova, refer to the paper by Jarno Holstein[1]. To learn about the evolution of the standard library of Meta-Casanova and its applications, refer to the paper by Louis van der Burg[2].

For more information on Meta-Casanova, see the Meta-Casanova paper at `https://github.com/vs-team/Papers` or browse the freely-available source code at `https://github.com/vs-team/metacasanova`.

Further work

In the immediate future there will be three bachelor theses on Meta-Casanova. Jarno Holstein will write his bachelors thesis on the implementation of the type-checker for the type-system with the extensions described in this paper. Louis van der Burg will write his on the implementation and design of the Meta-Casanova library in the Meta-Casanova language. Finally, I will write mine on the implementation of a reliable code-generator.

In the not-so-immediate future there may be a possibility of full dependent types. Dependent types are types that depend on terms, and are currently not supported by Meta-Casanova. An example for this is the constant-length array, where the length of the array is embedded into the type at compile-time.

A future inclusion of dependent types is possible. It would mean that the type namespace will be merged into the module namespace. This won't break backward compatibility, since all identifiers in Meta-Casanova need to be unique.

Although dependent types complement the TypeFunc and Module system, it does not have the priority at the moment. The possibility for dependent types is therefore left open for future extensions.

References

[1] Jarno Holstein. Syntactic Parsing in F# Using Monadic Parsers. 2016.

[2] Louis van der Burg. Adventures in Meta-Casanova. 2016.

[3] Bryan O'Sullivan, Don Stewart, and John Goerzen. Real World Haskell. `http://book.realworldhaskell.org`. O'Reilly, 2008.

[4] Benjamin C. Pierce. Types and Programming Languages. The MIT Press, 2002.